Information and Computation 214 (2012) 86–111

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Featherweight Jigsaw — Replacing inheritance by composition in Java-like languages ☆

Giovanni Lagorio *, Marco Servetto, Elena Zucca

DISI, University of Genova, v. Dodecaneso 35, 16146 Genova, Italy

ARTICLE INFO

Article history:
Received 22 December 2009
Revised 29 July 2011
Available online 8 February 2012

ABSTRACT

We present FJIG, a simple calculus where basic building blocks are classes in the style of Featherweight Java, declaring fields, methods and one constructor. However, inheritance has been generalized to the much more flexible notion originally proposed in Bracha's Jigsaw framework. That is, classes play also the role of modules, that can be composed by a rich set of operators, all of which can be expressed by a minimal core. Fields and methods can be declared of four different kinds (*abstract, virtual, frozen, local*) determining how they are affected by the operators.

We keep the nominal approach of Java-like languages, that is, types are class names. However, a class is not necessarily a structural subtype of any class used in its defining expression. While this allows a more flexible reuse, it may prevent the (generalized) inheritance relation from being a subtyping relation. So, the required subtyping relations among classes are declared by the programmer and checked by the type system.

The calculus allows the encoding of a large variety of different mechanisms for software composition in class-based languages, including standard inheritance, mixin classes, traits and hiding. Hence, FJIG can be used as a unifying framework for analyzing existing mechanisms and proposing new extensions.

We provide two different semantics of an FJIG program: *flattening* and *direct* semantics. The difference is analogous to that between two intuitive models to understand inheritance: the former where inherited methods are copied into heir classes, and the latter where member lookup is performed by ascending the inheritance chain. Here we address equivalence of these two views for a more sophisticated composition mechanism.

0. Introduction

Jigsaw is a framework for modular composition largely independent of the underlying language, designed by Gilad Bracha in his seminal thesis [1], and then formalized by a minimal set of operators in module calculi such as [2,3]. In this paper, we define an instantiation of Jigsaw, called Featherweight Jigsaw (FJIG for short), where basic building blocks are classes in the style of Java-like languages. That is, classes are collections of fields, methods and constructors, that can be instantiated to create objects; also, class names are used as types (nominal typing).

☆ This document is a collaborative effort and has been partially supported by MIUR DISCO — Distribution, Interaction, Specification, Composition for Object Systems.

* Corresponding author.
E-mail addresses: lagorio@disi.unige.it (G. Lagorio), servetto@disi.unige.it (M. Servetto), zucca@disi.unige.it (E. Zucca).

0890-5401/$ – see front matter
doi:10.1016/j.ic.2012.02.004

The motivation for this work is that, even though Jigsaw has been proposed a long time ago and since then it has been greatly influential,[1] its design has been never fully exploited in the context of Java-like languages, as recently pointed out as an open question [12]. Here, we provide a foundational answer to this question, by defining a core language which, however, embodies the key features of Java-like languages, in the same spirit of Featherweight Java [13] (FJ for short). Indeed, formally, a basic class of FJIG looks very much like a class in FJ. However, standard inheritance has been replaced by the much more flexible (module) composition, that is, by the rich set of operators of the Jigsaw framework.

Instantiating Jigsaw on Java-like languages poses some non trivial design problems. Just to mention one (others are discussed in Section 1), we keep the nominal approach of Java-like languages, that is, types are class names. However, a class is not necessarily a structural subtype of any class used in its defining expression. While this allows a more flexible reuse, it may prevent the (generalized) inheritance relation from being a subtyping relation. So, the required subtyping relations among classes are declared by the programmer and checked by the type system.

Another challenging issue is the generalization to FJIG of two intuitive models to understand inheritance: one where inherited methods are copied into heir classes, and the other one where member lookup is performed by ascending the inheritance chain. We address the equivalence of these two views for a much more sophisticated composition mechanism. Formally, we provide two different semantics for an FJIG program: *flattening* semantics, that is, by translation into a program where all composition operators have been performed, and *direct* semantics, that is, by formalizing a dynamic look-up procedure.

The paper is organized as follows. Section 1 provides an informal introduction to FJIG by using a sugared surface syntax. Section 2 introduces a lower level syntax and defines flattening semantics. Section 3 defines the type system and states its soundness. Section 4 defines direct semantics of FJIG and states the equivalence between the two semantics. In the Conclusion, we summarize the contribution of the paper and briefly discuss related and further work.

This paper is an improved and extended version, including full proofs, of [14,15].

1. An informal introduction

In this section we illustrate the main features of FJIG by using a sugared surface syntax, given in Fig. 1. We assume infinite sets of *class names* C, (*member*) *names* N, and *variables* x, including the special variable `this`. We use the bar notation for sequences, e.g., $\overline{\mu}$ is a metavariable for sequences $\mu_1 \dots \mu_n$.

This syntax is designed to keep a Java-like flavour as much as possible. In the next section we will use a lower-level representation, which allows us to formalize the semantics in a simpler and natural way.

A program consists of a sequence of *class declarations*, consisting of an optional `abstract` modifier, a class name and a class expression. Class expressions are basic classes, class names, or are inductively constructed by a set of composition operators. A basic class consists of a sequence of supertypes, a sequence of field declarations, a constructor declaration, and a sequence of method declarations.

We will first revise Jigsaw features in the context of FJIG, then discuss some issues that are specific to the instantiation on Java-like languages.

1.1. Basic classes

Jigsaw is a programming paradigm based on (module) composition, where a basic module (in our case, a class) is a collection of components (in our case, members), which can be of four different kinds, indicated by a modifier: `abstract`, `virtual`, `frozen`, and `local`. A method has no body if and only if its modifier is `abstract`. The meaning of modifiers is as follows:

- An `abstract` member has no definition, and is expected to be defined later when composing the class with others.
- A `virtual` or `frozen` member has a definition, which can be changed by using the composition operators. However, the redefinition of a `frozen` member does not affect the other members, that still refer to its original definition.
- Finally, as the name suggests, a `local` member cannot be selected by a client[2], and is not affected by composition operators, hence its definition cannot be changed.

We assume by default (hence omit) the modifier `frozen` for fields and `virtual` for methods. A class having at least one `abstract` member must be declared `abstract`.

The following example shows two basic classes.[3]

[1] Just to mention two different research areas, Jigsaw principles are present in work on extending the ML module system with mutually recursive modules [4–6], and Jigsaw operators already included those later used in mixin classes and traits [7–11].

[2] Note the difference with `private` modifier in Java, which allows client selection when clients are of the same class, see more details later in this paper.

[3] To write more readable examples, we assume that the primitive type `int` and its operations are available.

p	$::=$	$\overline{cd}$	program
cd	$::=$	$cmod$ `class` C CE	class declaration
$cmod$	$::=$	`abstract` $\mid \epsilon$	class modifier
CE	$::=$		class expression
		B	basic class
		$\mid C$	class name
		$\mid CE_1$ `merge` CE_2	merge
		$\mid CE_1$ `override` CE_2	override
		$\mid$ `rename` N `to` N' `in` CE	rename
		$\mid$ `restrict` N `in` CE	restrict
		$\mid$ `hide` N `in` CE	hide
		$\mid \ldots$	
		$\mid CE[kh\{\texttt{super}(\bar{e})\}]$	constructor wrapper
N	$::=$	$F \mid M$	member name
kh	$::=$	`constructor`$(\overline{C\ x})$	constructor header
B	$::=$	$I\{\overline{\varphi}\ \kappa\ \overline{\mu}\}$	basic class
I	$::=$	`implements` $\overline{C}$	supertypes
φ	$::=$	$mod\ C\ F;$	field
κ	$::=$	$kh\{\overline{F = e}\}$	constructor
μ	$::=$	$mod\ C\ M\ (\overline{C\ x})\{\texttt{return}\ e;\}$	
		$\mid$ `abstract` $C\ M(\overline{C\ x});$	method
mod	$::=$	`abstract` $\mid$ `virtual` $\mid$ `frozen` $\mid$ `local`	member modifier
e	$::=$		expression
		x	variable
		$\mid e.F$	client field access
		$\mid e.M(\bar{e})$	client method invocation
		$\mid F$	internal field access
		$\mid M(\bar{e})$	internal method invocation
		$\mid$ `new` $C(\bar{e})$	object creation

Fig. 1. FJIG (surface) syntax.

```
abstract class A {
  abstract int M1();
  int M2() { return M1() + M3(); }
  local int M3() { return 1; }
}
abstract class B {
  abstract int M2();
  frozen int M1() { return 1 + M2(); }
}
```

These two classes are abstract (hence cannot be instantiated).

1.2. Merge and override operators

A concrete class can be obtained by applying the `merge` operator as follows:

```
class C
  A merge B
```

This declaration is equivalent to the following:

```
class C {
  frozen int M1() { return 1 + M2(); }
  int M2() { return M1() + M3(); }
  local int M3() { return 1; }
}
```

Conflicting definitions for the same (non-local) member are not permitted, whereas `abstract` members with the same name are shared. Members can be selected by client code unless they are `local`, that is, we can write, e.g., `new C().M2()` but not `new C().M3()`. To show the difference between `virtual` and `frozen` members, in the next ex-

amples we use the `override` operator, a variant of `merge` where conflicts are allowed and the left argument has the precedence.

```
class D1
  { int M2() { return 2; } } override C
```

This declaration is equivalent to the following:

```
class D1 {
  frozen int M1() { return 1 + M2(); }
  int M2() { return 2; }
  local int M3() { return 1; }
}
```

An invocation `new D1().M2()` will evaluate to `2`, and an invocation `new D1().M1()` to `3`. On the other hand, in the case of this declaration:

```
class D2
  { int M1() { return 3; } } override C
```

which is equivalent to the following:

```
class D2 {
  int M1() { return 3;}
  local int M1_old() { return 1 + M2(); }
  int M2() { return M1_old() + M3(); }
  local int M3() { return 1; }
}
```

an invocation `new D2().M1()` will evaluate to `3`, *but* an invocation `new D2().M2()` will not terminate, since the internal invocation `M1()` in the body of `M2()` still refers to the old definition.

1.3. Client and internal member selection

In a programming paradigm based on module composition, a member can be either selected by a client, or used by other members inside the module itself. Correspondingly, in FJIG we distinguish between *client* field-accesses/method-invocations, which specify a receiver, and *internal* field-accesses/method-invocations, whose implicit receiver is the current object. Note that a client method invocation $e.M(\ldots)$ has a different meaning w.r.t. an internal method invocation $M(\ldots)$, even in the case the receiver expression e denotes an object of the same class (that is, internal selection *does not* correspond to selection of `private` members as in, e.g., Java). The following example[4] shows this difference:

```
class CA
  { int M(){ return 1; } }
  merge
  implements CA{
    int K(){ return
      new CA().M()
      + this.M()
    //+ M()
    ;}
  }
```

This declaration is equivalent to the following:

```
class CA implements CA {
    int M(){ return 1; }
    int K(){ return
      new CA().M()
      + this.M()
    //+ M()
    ;}
  }
```

[4] The supertype declaration `implements CA` is needed to type the `this` occurrence, as will be explained in more detail later on.

In the body of method K, the first expression, `new CA().M()`, is intuitively correct; the second one is correct because the supertype CA has been declared, thus `this` is an expression of type CA. Finally, the last (commented) invocation is not correct since the (unnamed) second basic class does not contain a declaration for a method named M.

Moreover internal field accesses and method invocations are affected by operators. For instance, consider the following class, where we use the operator `rename`, which changes the name of a member.

```
class E
  (rename M1 to M4 in {
    int M1() { return 1; }
    int M2() { return M1(); }
    int M3() { return new E().M1(); }
  })
  merge
  { int M1() { return 3; } }
```

This declaration is equivalent to the following:

```
class E {
  int M4() { return 1; }
  int M2() { return M4(); }
  int M3() { return new E().M1(); }
  int M1() { return 3; }
  }
```

An invocation `new E().M2()` returns `1`, since the internal invocation in the body of M2 refers to the method now called M4. However, an invocation `new E().M3()` returns `3`, since the client invocation in the body of M3 refers to method M1 in E.

Other operators of the Jigsaw framework, besides those above, are `restrict`, which eliminates the definition for a member,[5] and `hide`, which makes a member local. We refer to [1] and [3] for more details. All these operators and many others can be easily encoded (see [3]) by using a minimal set of *primitive* operators: *sum*, *reduct*, and *freeze*, which will be formally defined in next section. Sum is a low-level form of merge where the two arguments are required to have *exactly* the same constructor parameters. Reduct is a low-level form of renaming that replaces independently member names and their occurrences in method bodies, via two renamings which are two finite maps over names. The *freeze* operator makes a virtual member frozen.

We discuss now the issues specific to the instantiation on Java-like classes.

1.4. Fields and constructors

It turns out that the above modifiers can be smoothly applied to fields as well, with analogous meaning, as shown by the following example which also illustrates how constructors work.

```
class A1 {
  abstract int F1;
  virtual int F2;
  int F3;
  constructor(int x) { F2 = x; F3 = x; }
  int M() { return F2 + F3; }
}
class C1 {
  int F1;
  int F2;
  int F3;
  constructor(int x) {
    F1 = x + 1;
    F2 = x + 1;
    F3 = x + 1; }
} override A1
```

[5] Indeed, CE_1 `override` $CE_2 = CE_1$ `merge restrict` N_1 in ... `restrict` N_k in CE_2 where $N_1, \ldots, N_k$ are the common members.

A basic class defines one[6] constructor which specifies a sequence of parameters and a sequence of initialization expressions, one for each non-abstract field. We assume a default constructor with no parameters for classes having no fields. Note the difference with FJ, where the class constructor has a canonical form (parameters exactly correspond to fields). This would be inadequate in our framework since object layout must be hidden to clients.

In order to be composed by merge/overriding, two classes should provide a constructor with the same parameter list (if it is not the case, a *constructor wrapper* can be inserted, see the last example of this section), and the effect is that the resulting class provides a constructor with the same parameter list, that executes both the original constructors.

An instance of class `C1` has four[7] fields (`A1.F3`, `C1.F1`, `C1.F2`, `C1.F3`), and an invocation `new C1(5).M()` will return `11`, since `F2` in the body of `M` refers to the field declared in `C1` (initialized with `5+1`), while `F3` refers to the field declared in `A1` (initialized with `5`). Indeed, `F3` is frozen in (the basic class defining) A1, and, exactly as for methods, this means that references to `F1` in the code of this basic class will always refer to this declaration, even in case this code is inherited in a context where `F3` is overridden, as in this example. In other words, for frozen fields and methods the overriding operator has the effect of *hiding* the old version, as it happens in Java for static methods and fields (which are, following our terminology, all frozen).

Classes composed by merge/overriding can share the same field, provided it is abstract in all except (at most) one. Note that this corresponds to *sharing* fields as in, e.g., [16]; however, in our framework we do not need an ad-hoc notion.

Finally, note that a `super` mechanism, allowing to refer to an overridden member, such as `A1.F2`, can be encoded in Jigsaw, notably by renaming, as shown in [17].

1.5. Inheritance and subtyping

Since our aim is to instantiate the Jigsaw framework on a Java-like language, we keep a nominal approach, that is, types are class names. However, subtyping *does not* coincide with the generalized inheritance relation, since some of the composition operators (e.g., renaming) do not preserve structural subtyping. Instead, desired subtyping relations must be explicitly written by the programmer, by declaring a set $C_1 \dots C_n$ of *supertypes*, introduced by the keyword `implements`, in a basic class, as illustrated by the example below.

```
class CC {
   abstract int m1();
   abstract int m2();
}
class DD implements CC {
  abstract int m1();
  int m2() { return 1 + this.m1(); }
  CC m() { return this; }
}
```

In this way we can return `this` as result of method `m`. The type system checks, for each C_i, that the subtyping relation can be safely assumed, that is, members of C_i are members of the basic class as well.[8] This check is analogous to that on implemented interfaces in Java. For instance, removing method `m1` from `DD` would make the example ill-typed. Note that, differently from Java, where they are implicitly inherited, abstract members must be declared as well, so that class types can be computed in isolation.

As an abbreviation, we omit the `implements` keyword if the sequence of supertypes is empty.

To conclude this section, we show a more significant example, where we also assume to have the type `void` and some statements in the syntax.

The following class `DBSerializable`, an example of the pattern *template method* [18], contains the method `saveToDB`, which writes the object's serialized representation onto a database. While the behaviour of `saveToDB` is fixed, the details on how to open the connection are left unspecified, and the implementation of the method `serialize` can be changed.[9] This is reflected by the method modifiers. Class `DBConnection` is a given library class.

```
abstract class DBSerializable {
  abstract DBConnection openConnection();
  virtual void serialize(DBConnection c) {}
```

[6] Since, as in FJ and differently from Java, overloading is not allowed. An extension allowing overloading for methods should be modelled, exactly as in FJ, by introducing two different languages, one corresponding to source code and the other corresponding to (annotated) bytecode. Moreover, FJig operators currently handling method names should either handle method signatures (name + argument types) or uniformly act on all overloaded versions of a method. Allowing overloading for constructors, instead, requires non trivial design choices about the merge and the constructor wrapper operators.

[7] In the low-level representation introduced in next section, the overridden field `A1.F2` is kept as "garbage" which is never accessed.

[8] Formally, that the *class type* of the basic class is subtype of the class type of C_i, see rule (STRUCTURAL-SUB) in Fig. 9.

[9] This method could be declared abstract as well.

```
  frozen void saveToDB() {
    DBConnection connection = openConnection();
    // ...
    serialize(connection);
    connection.close();
  }
}
```

Suppose we want to specialize the class `DBSerializable` for the DB server MySQL. We can create this specialization, called `MySQLSerializable`, in two steps: first, we provide an implementation of `openConnection` with the specific code for MySQL, then we *hide* it, since clients of `MySQLSerializable` should never invoke this method directly. We start by defining an auxiliary class `_MySQLSerializable`, merging `DBSerializable` with an anonymous basic class:

```
class _MySQLSerializable
    DBSerializable[ constructor(String cs) {
                        super()
                    } ]
     merge
    { local String connectionString;
      constructor(String cs) {
          connectionString = cs;
      }
      virtual DBConnection openConnection() {
        /* ... use connectionString ... */}
    }
```

Note the use of the constructor wrapper: the constructor of the anonymous basic class has a `String` parameter, whereas that of the class `DBSerializable`, which has no fields, is the default (parameterless) constructor. Hence, a constructor wrapper is inserted, so that the classes we are merging have both a constructor with the same parameters. This allows writing creation expressions like

```
new _MySQLSerializable("someConnectionString")}
```

As mentioned before, the class `_MySQLSerializable` provides, along with the method `saveToDB`, the method `openConnection` which we can hide as follows:

```
class MySQLSerializable
  hide openConnection in _MySQLSerializable
```

Consider now the following class `Person`, providing a method, named `write`, to serialize its objects to a database:

```
class Person {  // ...
  frozen void write(DBConnection c) {
     /* serializes the data on c*/}
}
```

Although the inherited method `DBSerializable.saveToDB` writes the data by invoking the method `serialize` and not `write`, using the class `Person` with `MySQLSerializable` is not a problem, since we can rename the method before merging the two classes:

```
class MySQLSerializablePerson
  hide serialize in
    (rename write to serialize in Person)
      [constructor(String cs){super()}]
    override MySQLSerializable
```

2. FJIG calculus

In this section we formally define the (flattening) semantics of FJIG. To this aim, we use a different representation for basic classes w.r.t. the surface syntax given in Fig. 1. That is, instead of having explicit modifiers, their semantics is encoded by distinguishing between *external* and *internal* member names. Internal names are used to refer to members inside the class itself, whereas external names are used in class composition via operators and in selection of members by clients.

Hence, basic classes have shape $[\iota \mid o \mid \rho]$ where ι is an *input map* from internal to external names, o is an *output map* from external to internal names, and ρ is a *local part* which contains the actual code. Intuitively, ι maps required internal names to external names imported from other classes, and o maps exported external names to internal names with associated definitions. For instance, the basic class defining C shown in the previous section

```
class C {
  frozen int M1() { return 1 + M2(); }
  int M2() { return M1() + M3(); }
  local int M3() { return 1; }
}
```

is represented as follows:

$$
\begin{array}{l}
[m_2 : (() \to \texttt{int}) \mapsto M_2 | M_1 : (() \to \texttt{int}) \mapsto m_1', M_2 : (() \to \texttt{int}) \mapsto m_2', |\rho] \\
\rho = \texttt{Object}\{ \\
\quad ()\{\} \\
\quad \texttt{int}\, m_1'()\{\texttt{return}\, 1 + m_2();\} \\
\quad \texttt{int}\, m_2'()\{\texttt{return}\, m_1'() + m_3'();\} \\
\quad \texttt{int}\, m_3'()\{\texttt{return}\, 1;\} \\
\}
\end{array}
$$

In general, to encode a basic class of the surface language in the calculus, we need for each member name N of type T (at most) a corresponding external name N and (at most) two internal names n, n', depending on the member kind, as detailed below. Client references to N are unaffected, whereas internal references are translated according to the member kind:

- if N is abstract, then there is an association $n : T \mapsto N$ in the input map, and internal references are translated by n,
- if N is virtual, as M2 in the example, then there is an association $n : T \mapsto N$ in the input map, an association $N : T \mapsto n'$ in the output map, a definition for n' in ρ, and internal references are translated by n,
- if N is frozen, then there is an association $N : T \mapsto n'$ in the output map, a definition for n' in ρ, and internal references are translated by n',
- if N is local, then there is no corresponding external name, and there is a definition for n' in ρ, and internal references are translated by n'.

Inside constructor bodies, a field name F on the left-hand side is always translated by the internal name f' (and internal member selection is forbidden in the initialization expressions).

We could have alternatively expressed the semantics directly on the surface language, getting a more FJ-like flavour, as we do in another paper [19] for a version of FJig with nested classes and only abstract or virtual members. However, the representation with i/o maps has some advantages: a clean distinction between internal names, which can be α-renamed, and external names, as in the tradition of module calculi, see [2,3]; operators can be modeled in a uniform way, whereas expressing the semantics on the surface language requires a case analysis on the kind (abstract, virtual, frozen and local) of members; finally, the representation with i/o maps corresponds to what we expect from an implementation, that is, that code contains internal references to external names, so that renaming does not require to modify code, but just to update a pointer, as formalized by map composition in rule (REDUCT).

The syntax of the calculus is given in Fig. 2. Besides class names, (external) names and variables, we assume an infinite set of *internal* (*member*) *names* n. A program consists of a sequence of *class declarations* (class name and class expression), as in FJ. We assume that no class is declared twice and order is immaterial, hence we can write $p(C)$ for the class expression associated with C. Class expressions CE are basic classes B, class names C, or are inductively constructed by a set of composition operators. Let us say that C "inherits from" C' if the class expression associated with C contains a subterm C', or, transitively, C'' which inherits from C'. In a well-formed program, we require this generalized inheritance relation to be acyclic, exactly as for standard inheritance.

Input and output maps are represented as sequences of pairs where the first element has a type annotation. In an input map, internal names which are mapped to the same external name are required to have the same annotation, whereas this is not required in output names, that is, the same member can be exported under different names with different types, see the type system in next section. This is illustrated by the following example, where class C_2 is obtained by applying the reduct operator to C_1, thus exporting method m_1' twice, as M_1 of type $() \to C_1$ and as M_2 of type $() \to \texttt{Object}$, respectively.

$$
\begin{array}{l}
C_1 = [|M_1 : (() \to C_1) \mapsto m_1' | \texttt{Object}\{()\{\} C_1\, m_1'()\{\texttt{return}\, m_1'();\}\}] \\
C_2 = {}_{|}C_{1\,|M_1:(()\to C_1)\mapsto M_1:(()\to C_1), M_2:(()\to \texttt{Object})\mapsto M_1:(()\to C_1)}
\end{array}
$$

p	$::=$	$\overline{cd}$	program
cd	$::=$	$C \mapsto CE$	class declaration
CE	$::=$	$B \mid C \mid$	class expression
		$CE_1 + CE_2$	sum
		$\mid {}_{\sigma^\iota}\mid CE \mid_{\sigma^o}$	reduct
		$\mid freeze_N CE$	freeze
		$\mid CE[(\overline{C\ x})\{\bar{e}\}]$	constructor wrapper
σ	$::=$	$\overline{N : T \mapsto N' : T'}, \overline{\_ \mapsto N : T}$	renaming
N	$::=$	$F \mid M$	external member name
T	$::=$	$C \mid \bar{C} \to C$	member type
B	$::=$	$[\iota \mid o \mid \rho]$	basic class
ι	$::=$	$\overline{n : T \mapsto N}$	input map
o	$::=$	$\overline{N : T \mapsto n}$	output map
n	$::=$	$f \mid m$	internal member name
ρ	$::=$	$I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}$	local part
I	$::=$	$\bar{C}$	supertypes
φ	$::=$	$C\ f;$	field
κ	$::=$	$(\overline{C\ x})\{\phi\}$	constructor
ϕ	$::=$	$\overline{f = e}$	constructor body
μ	$::=$	$C\ m(\overline{C\ x})\{\texttt{return}\ e;\}$	method
e	$::=$	$x \mid e.F \mid e.M(\bar{e}) \mid f \mid m(\bar{e}) \mid \texttt{new}\ C(\bar{e})$	expression
		$\mid [\overline{\mu}; v \mid e]$	block
		$\mid C(\phi)$	(pre-)object
v, v^C	$::=$	$C(\overline{f = v})$	value (object)

Fig. 2. Syntax.

Renamings σ are maps from (annotated) external names into (annotated) external names, represented as sequences of pairs; pairs of form $\_ \mapsto N : T$ are used to represent non-surjective maps. A non-surjective renaming allows one to remove an external name from the output map, or to add an external name to the input map.

We omit some keywords w.r.t. the surface syntax, and expressions include *runtime expressions*, that is, (pre-)objects and blocks.

We denote by *dom* and *img* the domain and image of a map, respectively. Given a basic class $[\iota \mid o \mid \rho]$, with $\rho = I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}$, we denote by $dom(\overline{\mu})$ and $dom(\overline{\varphi})$ the sets of internal names declared in $\overline{\mu}$ and $\overline{\varphi}$, respectively, which are assumed to be disjoint. The union of these two sets, denoted by $dom(\rho)$, is the set of *local* names.

In a well-formed basic class, internal names of local and abstract/virtual members must be distinct, that is, $dom(\iota) \cap dom(\rho) = \emptyset$. Moreover, $img(o) \subseteq dom(\rho)$, and, denoting by $names(e)$ the set of internal names in an expression e, $names(e) \subseteq dom(\iota) \cup dom(\rho)$ for each method body e.

We describe now the two kinds of runtime expressions introduced in the calculus.

Expressions of form $C(\overline{f = e})$, called pre-objects of class C, where for each field there is an initialization expression, model intermediate steps in the evaluation of a constructor. Note the difference with the form $\texttt{new}\ C(\bar{e})$, which denotes a constructor invocation, whereas in FJ objects can be identified with object creation expressions where arguments are values. As already noted, in FJ it is possible, and convenient, to take this simple and nice solution, since the structure of the instances of a class is globally visible to the whole program. In FJIG, instead, object layout must be hidden to clients, hence constructor parameters have no a priori relation with fields.

Values of the calculus are *objects*, that is, pre-objects where all expressions are (in turn) values. We use both v^C and v as metavariables for values of class C, the latter when the class is not relevant.

Moreover, runtime expressions also include *block* expressions $[\overline{\mu}; v \mid e]$, modeling execution of e with method internal names bound in $\overline{\mu}$ and field internal names in the current object v. Hence, denoting by $dom(v)$ the set $\{f_1, \ldots, f_n\}$ if $v = C\ (f_1 = v_1 \ldots f_n = v_n)$, a block expression is well formed only if $names(e) \subseteq dom(\overline{\mu}) \cup dom(v)$ and the sets $dom(\overline{\mu})$, $dom(v)$ are disjoint. Note that this implies $names([\overline{\mu}; v \mid e]) = \emptyset$.

The semantics of an expression e in the context of a program p can be defined in two different ways.

The former, which we call *flattening semantics* and illustrate in this section, is given in two steps. First, we define a flattening relation $p \longrightarrow p'$ which reduces a program, in some steps, to a *flat* program, that is, a program where every class is basic. To this end, operators are performed and the occurrences of class names are replaced by their defining expressions. Then, we define a relation $e \longrightarrow_p e'$ which models reduction of an expression in the context of a flat program p. Note that in this case dynamic look-up is always trivial, that is, a class member (e.g., a method) can be always found in the class of

$$(\text{CDEC1})\ \frac{CE \longrightarrow CE'}{(p, C \mapsto CE) \longrightarrow (p, C \mapsto CE')} \qquad (\text{CDEC2})\ \frac{}{(p, C \mapsto B) \longrightarrow (p[B/C], C \mapsto B)}$$

$$(\mathcal{CE})\ \frac{CE \longrightarrow CE'}{\mathcal{CE}\{CE\} \longrightarrow \mathcal{CE}\{CE'\}}$$

$$(\text{SUM})\ \frac{}{[\iota_1 \mid o_1 \mid \rho_1] + [\iota_2 \mid o_2 \mid \rho_2] \longrightarrow [\iota_1, \iota_2 \mid o_1, o_2 \mid \rho]} \begin{array}{l} \rho_i = \overline{C}_i\ \{\overline{\varphi}_i\ (\overline{C\,x})\{\phi_i\}\ \overline{\mu}_i\},\ i \in \{1,2\} \\ \rho = \overline{C}_1, \overline{C}_2\ \{\overline{\varphi}_1, \overline{\varphi}_2\ (\overline{C\,x})\{\phi_1, \phi_2\}\ \overline{\mu}_1, \overline{\mu}_2\} \end{array}$$

$$(\text{REDUCT})\ \frac{}{{}^{\sigma^\iota|}[\iota \mid o \mid \rho]_{|\sigma^o} \longrightarrow [\sigma^\iota \circ \iota \mid o \circ \sigma^o \mid \rho]}$$

$$(\text{FREEZE})\ \frac{}{\mathit{freeze}_N[\iota, n_1 : T \mapsto N \ldots n_k : T \mapsto N \mid o \mid \rho] \longrightarrow [\iota \mid o \mid \rho[n'/n_1] \ldots [n'/n_k]]} \begin{array}{l} n' = o(N) \\ N \notin \mathit{img}(\iota) \end{array}$$

$$(\text{CONS-WRAPPER})\ \frac{}{[\iota \mid o \mid \rho][(\overline{D\,y})\{\bar{e}\}] \longrightarrow [\iota \mid o \mid \rho']} \begin{array}{l} \bar{x} = x_1 \ldots x_n \\ \rho = \overline{C}\ \{\overline{\varphi}\ (C_1\,x_1 \ldots C_n\,x_n)\{\phi\}\ \overline{\mu}\} \\ \rho' = \overline{C}\ \{\overline{\varphi}\ (\overline{Dy})\{\phi[\bar{e}/\bar{x}]\}\ \overline{\mu}\} \end{array}$$

$$(\mathcal{E})\ \frac{e \longrightarrow_p e'}{\mathcal{E}\{e\} \longrightarrow_p \mathcal{E}\{e'\}} \qquad (\text{CLIENT-FIELD})\ \frac{}{v^C.F \longrightarrow_p [\overline{\mu}; v^C \mid f]} \begin{array}{l} p(C) = [\iota \mid o \mid \overline{C}\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}] \\ o(F) = f \end{array}$$

$$(\text{CLIENT-INVK})\ \frac{}{v^C.M(\bar{v}) \longrightarrow_p [\overline{\mu}; v^C \mid m(\bar{v})]} \begin{array}{l} p(C) = [\iota \mid o \mid \overline{C}\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}] \\ o(M) = m \end{array}$$

$$(\text{INT-FIELD})\ \frac{}{[\overline{\mu}; v \mid \mathcal{E}\{f\}] \longrightarrow_p [\overline{\mu}; v \mid \mathcal{E}\{v_i\}]} \begin{array}{l} \mathit{HB}(\mathcal{E}) = \emptyset \\ v = C(f_1 = v_1 \ldots f_n = v_n) \\ f = f_i \end{array}$$

$$(\text{INT-INVK})\ \frac{}{[\overline{\mu}; v^C \mid \mathcal{E}\{m(\bar{v})\}] \longrightarrow_p [\overline{\mu}; v^C \mid \mathcal{E}\{e[\bar{v}/\bar{x}][v^C/\texttt{this}]\}]} \begin{array}{l} \mathit{HB}(\mathcal{E}) = \emptyset \\ \overline{\mu}(m) = \langle \bar{x}, e \rangle \end{array}$$

$$(\text{OBJ-CREATION})\ \frac{}{\texttt{new}\ C(\bar{v}) \longrightarrow_p C(\phi[\bar{v}/\bar{x}])} \begin{array}{l} p(C) = [\emptyset \mid o \mid \rho] \\ \rho = \overline{C}\ \{\overline{\varphi}\ (C_1\ x_1 \ldots C_n\ x_n)\{\phi\}\ \overline{\mu}\} \\ \bar{x} = x_1 \ldots x_n \end{array}$$

$$(\text{EXIT-BLOCK})\ \frac{}{[\overline{\mu}; v \mid e] \longrightarrow_p e}\ \mathit{names}(e) = \emptyset$$

Fig. 3. Flattening semantics.

the receiver. In next section, we define an alternative *direct* semantics, where expressions are reduced in the context of non flat programs, hence where dynamic look-up is non trivial.

Flattening rules are defined in the top section of Fig. 3.

The first two rules define reduction steps of programs, which can be obtained either by reducing one of the class expressions, or, if some class C has already been reduced to a basic class B, by replacing by B all occurrences of C as class expression, formally:

$$C[B/C] = B$$

$$C'[B/C] = C' \qquad \text{if } C' \neq C$$

$$B'[B/C] = B'$$

$$(CE_1 + CE_2)[B/C] = (CE_1[B/C]) + (CE_2[B/C])$$

$$(\mathit{freeze}_N CE)[B/C] = \mathit{freeze}_N(CE[B/C])$$

$${}^{\sigma^\iota|}CE_{|\sigma^o}[B/C] = {}^{\sigma^\iota|}CE[B/C]_{|\sigma^o}$$

$$(CE[(\overline{C\,x})\{\phi\}])[B/C] = (CE[B/C])[(\overline{C\,x})\{\phi\}]$$

Set $\bar{x} = x_1 \dots x_n$, $\bar{v} = v_1 \dots v_n$.

$x_i[\bar{v}/\bar{x}]$	$= v_i$	for all $i \in 1..n$
$x[\bar{v}/\bar{x}]$	$= x$	if $x \neq x_i$ for all $i \in 1..n$
$e.F[\bar{v}/\bar{x}]$	$= e[\bar{v}/\bar{x}].F$	
$e.M(\bar{e})[\bar{v}/\bar{x}]$	$= e[\bar{v}/\bar{x}].M(\bar{e}[\bar{v}/\bar{x}])$	
$f[\bar{v}/\bar{x}]$	$= f$	
$m(\bar{e})[\bar{v}/\bar{x}]$	$= m(\bar{e}[\bar{v}/\bar{x}])$	
$\texttt{new}\ C(\bar{e})[\bar{v}/\bar{x}]$	$= \texttt{new}\ C(\bar{e}[\bar{v}/\bar{x}])$	
$[\overline{\mu}; v \mid e][\bar{v}/\bar{x}]$	$= [\overline{\mu}; v \mid e[\bar{v}/\bar{x}]]$	
$C(\phi)[\bar{v}/\bar{x}]$	$= C(\phi[\bar{v}/\bar{x}])$	
$(f_1 = e_1 \dots f_n = e_n)[\bar{v}/\bar{x}]$	$= f_1 = (e_1[\bar{v}/\bar{x}]) \dots f_n = (e_n[\bar{v}/\bar{x}])$	
$(e_1 \dots e_n)[\bar{v}/\bar{x}]$	$= e_1[\bar{v}/\bar{x}] \dots e_n[\bar{v}/\bar{x}]$	

Fig. 4. Multiple substitution.

The remaining rules define reduction steps of class expressions. The first rule is the standard contextual closure, where $\mathcal{CE}$ denotes a one hole context, and $\mathcal{CE}\{CE\}$ denotes the class expression obtained by filling the hole by CE.

Formally:

$$\mathcal{CE} ::= \square \mid \mathcal{CE} + CE \mid CE + \mathcal{CE} \mid {}_{\sigma^\iota}|\mathcal{CE}_{|\sigma^o} \mid \mathit{freeze}_N\, \mathcal{CE} \mid \mathcal{CE}[(\overline{C\ x})\{\bar{e}\}]$$

Rules for sum, reduct and freeze operators are essentially those given in [3].

Sum operation has the effect of gluing together two classes. The expression o_1, o_2 is well formed only if the two maps have disjoint domains, and analogously for other maps. Hence, rule (SUM) can only be applied when sets of internal names are disjoint ($(dom(\iota_1) \cup dom(\rho_1) \cap (dom(\iota_2) \cup dom(\rho_2)) = \emptyset$), as are sets of output names ($dom(o_1) \cap dom(o_2) = \emptyset$). The former implicit side condition can be always satisfied by an appropriate α-conversion, whereas the latter corresponds to a conflict that the programmer can only solve by an explicitly renaming (reduct operator). The sets of input names can have a non empty intersection and the resulting set of the input names of the sum is simply the union of them; this means that input members having the same name are shared. However, to ensure well-formedness of the resulting input map, they must have the same type. Finally, constructor parameters are required to be the same, in order both to get a commutative operator and to keep the calculus minimal; indeed, this can be always achieved by using the constructor wrapper operator.

In rule (REDUCT), new input and output names are chosen, modeled by $img(\sigma^\iota)$ and $dom(\sigma^o)$, respectively. Old input names are mapped in new input names by σ^ι, whereas new output names are mapped into old output names by σ^o. Input names can be shared or added, whereas output names can be duplicated or removed. The symbol $\circ$ denotes composition of maps, which is well formed only if type annotations are the same and the annotation of the new name is kept in the resulting map. That is: if ι contains $n : T \mapsto N$, then σ^ι should contain $N : T \mapsto N' : T'$, and $\sigma^\iota \circ \iota$ will contain $n : T' \mapsto N'$; if σ^o contains $N' : T' \mapsto N : T$, then o should contain $N : T \mapsto n$, and $o \circ \sigma^o$ will contain $N' : T' \mapsto n$.

In rule (FREEZE), associations from internal names into N are removed from the input map, and occurrences of these names in method bodies are replaced by the local name of the corresponding definition, thus eliminating any dependency on N. The second side condition ensures that we actually take *all* such names.

In rule (CONS-WRAPPING), n is the arity of the old constructor, and the body of the new constructor has n initialization expressions, as implicitly imposed by the well-formedness of multiple substitution $\bar{e}$ for $\bar{x}$ (see Fig. 4).

Reduction rules are given in the second section of Fig. 3.

The first rule is the standard contextual closure, where $\mathcal{E}$ denotes a one-hole context (see Fig. 5), and $\mathcal{E}\{e\}$ denotes the expression obtained by filling the hole by e.

Client field accesses and method invocations are reduced in two steps. First, they are reduced to a block where the current object is the receiver and the expression to be executed is the corresponding internal member selection on the name found in the receiver's class; moreover, methods found in the receiver's class are copied into the block and used for resolving further internal method invocations. Alternatively, the method body corresponding to an internal name could be again found in the basic class of the receiver; we choose this model because it can be better generalized to direct semantics, see the following section. Then, the following two rules can be applied.

An internal field access can only be reduced if it appears inside a block. In this case, it is replaced by the corresponding field of the current object. The first side condition says that the occurrence of f or m in the position denoted by the hole of the context $\mathcal{E}$ is free (that is, not captured by any binder around the hole), hence ensures that it is correctly bound to the current object in the first enclosing block. The standard formal definition of *HB* is in Fig. 6. For instance, in the expression $[\overline{\mu}; v \mid m(f, [\overline{\mu}'; v' \mid f])]$, the first occurrence of f denotes a field of the object v, whereas the second occurrence denotes a field of the object v'. Analogously, an internal method invocation is replaced by the corresponding body, found in $\overline{\mu}$, where parameters are replaced by arguments and `this` by the current object. We denote by $\overline{\mu}(m)$ the pair $\langle x_1 \dots x_n, e\rangle$ if $\overline{\mu}$ contains a (unique) method $C\ m(C_1\ x_1\ \dots C_n\ x_n)\{\texttt{return}\ e;\}$.

Note that there are two kinds of references to the current object in a method body: through the keyword `this` (which can appear in client member selection, or in a non-receiver position, e.g., `return this`), and through internal names.

$$\begin{array}{l}\mathcal{E} ::= \Box \mid \mathcal{E}.F \mid \mathcal{E}.M(\bar{e}) \mid e.M(\bar{e}, \mathcal{E}, \bar{e}') \\ \qquad \mid m(\bar{e}, \mathcal{E}, \bar{e}') \mid \texttt{new}\ C(\bar{e}, \mathcal{E}, \bar{e}') \\ \qquad \mid [\overline{\mu}; v \mid \mathcal{E}] \mid C(\overline{f = e}, f = \mathcal{E}, \overline{f = e'})\end{array}$$

Fig. 5. One hole context for expressions.

$$\begin{array}{ll}HB(\Box) & = \emptyset \\ HB(\mathcal{E}.F) & = HB(\mathcal{E}) \\ HB(\mathcal{E}.M(\bar{e})) & = HB(\mathcal{E}) \\ HB(e.M(\bar{e}, \mathcal{E}, \bar{e}')) & = HB(\mathcal{E}) \\ HB(m(\bar{e}, \mathcal{E}, \overline{e'})) & = HB(\mathcal{E}) \\ HB(\texttt{new}\ C(\bar{e}, \mathcal{E}, \bar{e}')) & = HB(\mathcal{E}) \\ HB([\overline{\mu}; C(\phi) \mid \mathcal{E}]) & = HB(\mathcal{E}) \cup dom(\overline{\mu}) \cup dom(\phi) \\ HB(C(\overline{f = e}, f = \mathcal{E}, \overline{f = e'})) & = HB(\mathcal{E})\end{array}$$

Fig. 6. Hole binder.

Whereas the former can be substituted at invocation time, as in FJ, the latter are modeled by a block, otherwise we would not be able to distinguish, among the objects of form v^C, those which actually refer to the original receiver of the invocation.

In rule (OBJ-CREATION), note that only classes where all members are frozen can be instantiated. This is a simplification: the execution model could be easily generalized to handle internal member selection on a virtual member by retrieving the input map as well in blocks, in rules (CLIENT-FIELD) and (CLIENT-INVK), and adding two reduction rules which, roughly, reduce such an internal field access/method invocation into the corresponding client member selection. We preferred to stick to an equivalent simpler model which, assuming that all classes have been frozen before being instantiated, avoids these redundant lookup steps.

Finally, in (EXIT-BLOCK), a block can be eliminated when the enclosed expression does no longer contain internal member selections, hence in particular when a value is obtained.

We conclude this section with an example of reduction on a trivial class `Count` where, for simplicity, we use integer literals and the sum on integers.

$$\begin{array}{l}\texttt{Count} = [\ \mid \texttt{incr} : () \to \texttt{Count} \mapsto m \mid \texttt{Object}\ \{\ (\texttt{int}\ x)\{f = x;\}\ \mu\ \texttt{int}\ f;\}] \\ \texttt{where}\ \mu = \texttt{Count}\ m()\{\texttt{return}\ \texttt{new}\ \texttt{Count}(f + 1);\};\end{array}$$

This class contains a field f that can be incremented by invoking the method `incr` (actually, the method returns a new object with the incremented field).

The following reduction, where trivial contextual closures have been omitted, shows that the expression `new Count(0).incr()` reduces to `Count(`$f = 1$`)`.

$$\begin{array}{ll}\texttt{new}\ \texttt{Count}(0).\texttt{incr}() \longrightarrow_p & (\text{OBJ-CREATION}) \\ \texttt{Count}(f = 0).\texttt{incr}() \longrightarrow_p & (\text{CLIENT-INVK}) \\ [\mu; \texttt{Count}(f = 0) \mid \texttt{new}\ \texttt{Count}(f + 1)] \longrightarrow_p & (\text{INT-FIELD}) \\ [\mu; \texttt{Count}(f = 0) \mid \texttt{new}\ \texttt{Count}(0 + 1)] \longrightarrow_p & (+) \\ [\mu; \texttt{Count}(f = 0) \mid \texttt{new}\ \texttt{Count}(1)] \longrightarrow_p & (\text{OBJ-CREATION}) \\ [\mu; \texttt{Count}(f = 0) \mid \texttt{Count}(f = 1)] \longrightarrow_p & (\text{EXIT-BLOCK}) \\ \texttt{Count}(f = 1) & \end{array}$$

Other examples illustrating flattening semantics (in comparison with direct semantics) will be provided in Section 4.

3. Type system

The type system uses four kinds of type environments, shown in Fig. 7.

A class type environment is a map from class names into class types. A class type is a 4-tuple consisting of input and output signatures, constructor type and supertypes.

Signatures are maps from external names into types.

We denote by $mtype(\Delta, C, N)$ the type of member named N in $\Delta(C)$. For virtual members, which have both an input and output type, this type is the (more specific) output type to provide a richer interface to clients. Indeed, internal references to a virtual member in C are typechecked assuming its input type, say T, whereas it could have an output type $T' \leqslant T$, which can be safely assumed by clients.

Δ	$::= \overline{C : CT}$	class type environment
CT	$::= [\Sigma^\iota; \Sigma^o; \overline{C}; I]$	class type
Γ	$::= \overline{n : T}$	internal type environment
Π	$::= \overline{x : C}$	parameter type environment
Σ	$::= \overline{N : T}$	signature
Δ^r	$::= \overline{C : \Gamma}$	runtime class type environment

Fig. 7. Type environments.

$$(\textsc{prog-t})\ \frac{\Delta \vdash CE_i : CT_i \quad \forall i \in 1..n \qquad \Delta \vdash C_i \leqslant I_i \text{ OK } \forall i \in 1..n}{\vdash C_1 \mapsto CE_1 \dots C_n \mapsto CE_n : \Delta}\ \begin{array}{l}\Delta = C_1 : CT_1 \dots C_n : CT_n \\ CT_i = [\_; \_; \_; I_i]\end{array}$$

$$(\textsc{cname-t})\ \frac{}{\Delta \vdash C : CT}\ \Delta(C) = CT$$

$$(\textsc{basic-t})\ \frac{\Delta; \Gamma^\iota, \Gamma^{\overline{\mu}}, \Gamma^{\overline{\varphi}}; I \vdash \overline{\mu} : \Gamma^{\overline{\mu}} \qquad \Delta; \Gamma^{\overline{\varphi}} \vdash \kappa : \overline{C}}{\Delta \vdash [\iota \mid o \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}] : [\Sigma^\iota; \Sigma^o; \overline{C}; I]}\ \begin{array}{l}\Delta \vdash \Sigma^o(N) \leqslant \Sigma^\iota(N)\ \forall N \in img(\iota) \cap dom(o) \\ \Delta \vdash (\Gamma^{\overline{\varphi}}, \Gamma^{\overline{\mu}})(o(N)) \leqslant \Sigma^o(N)\ \forall N \in dom(o)\end{array}$$

$$(\textsc{methods-t})\ \frac{\Delta; \Gamma; I \vdash \mu_i : T_i\ \forall\ i \in 1..n}{\Delta; \Gamma; I \vdash \overline{\mu} : \Gamma^{\overline{\mu}}}\ \begin{array}{l}\overline{\mu} = \mu_1 \dots \mu_n \\ \Gamma^{\overline{\mu}} = m_1 : T_1 \dots m_n : T_n\end{array}$$

$$(\textsc{method-t})\ \frac{\Delta; \Gamma; I; x_1 : C_1 \dots x_n : C_n \vdash e : C'}{\Delta; \Gamma; I \vdash C_0\ m(C_1\ x_1 \dots C_n\ x_n)\{\texttt{return}\ e;\} : C_1 \dots C_n \to C_0}\ \Delta \vdash C' \leqslant C_0$$

$$(\kappa\textsc{-t})\ \frac{\Delta; \emptyset; \emptyset; x_1 : C_1 \dots x_n : C_n \vdash e_i : C_i''\ \forall i \in 1..k}{\Delta; f_1 : C_1' \dots f_k : C_k' \vdash \kappa : C_1 \dots C_n}\ \begin{array}{l}\kappa = (C_1\ x_1 \dots C_n\ x_n)\{f_1 = e_1 \dots f_k = e_k\} \\ \Delta \vdash C_i'' \leqslant C_i'\ \forall i \in 1..k\end{array}$$

$$(\textsc{sum-t})\ \frac{\Delta \vdash CE_1 : [\Sigma_1^\iota; \Sigma_1^o; \overline{C}; I] \qquad \Delta \vdash CE_2 : [\Sigma_2^\iota; \Sigma_2^o; \overline{C}; I]}{\Delta \vdash CE_1 + CE_2 : [\Sigma_1^\iota, \Sigma_2^\iota; \Sigma_1^o, \Sigma_2^o; \overline{C}; I]}\ dom(\Sigma_1^o) \cap dom(\Sigma_2^o) = \emptyset$$

$$(\textsc{reduct-t})\ \frac{\Delta \vdash CE : [\Sigma_1^\iota; \Sigma_1^o; \overline{C}; I]}{\Delta \vdash {}_{\sigma^\iota|}CE_{|\sigma^o} : [\Sigma^\iota; \Sigma^o; \overline{C}; I]}\ \begin{array}{l}\Delta \vdash \sigma^\iota : \Sigma_1^\iota \to \Sigma^\iota \\ \Delta \vdash \sigma^o : \Sigma^o \to \Sigma_1^o\end{array}$$

$$(\textsc{freeze-t})\ \frac{\Delta \vdash CE : [\Sigma^\iota, N : T; \Sigma^o; \overline{C}; I]}{\Delta \vdash freeze_N CE : [\Sigma^\iota; \Sigma^o; \overline{C}; I]}\ N \in dom(\Sigma^o)$$

$$(\kappa\textsc{-wrapper-t})\ \frac{\Delta; \emptyset; \emptyset; x_1 : C_1 \dots x_n : C_n \vdash e_i : C_i''\ \forall i \in 1..k \qquad \Delta \vdash CE : [\Sigma^\iota; \Sigma^o; C_1' \dots C_k'; I]}{\Delta \vdash CE[(C_1\ x_1 \dots C_n\ x_n)\{e_1 \dots e_k\}] : [\Sigma^\iota; \Sigma^o; C_1 \dots C_n; I]}\ \begin{array}{l}\Delta \vdash C_i'' \leqslant C_i' \\ \forall i \in 1..k\end{array}$$

Fig. 8. Typing rules for programs and class expressions.

Internal type environments map internal names to types. Parameter type environments map variables (parameters) into class names. Finally, runtime class type environments map class names to internal type environments.

Typing rules in Fig. 8 define judgments $\vdash p : \Delta$ for programs and $\Delta \vdash CE : CT$ for class expressions.

In (PROG-T), a program has type Δ if each declared class C has type $\Delta(C)$ w.r.t. Δ, and, moreover, the declared subtyping relations can be safely assumed. The judgment $\Delta \vdash C \leqslant C_1 \dots C_n$ OK is an abbreviation for $\Delta \vdash C \leqslant C_i$ OK $\forall i \in 1..n$.

In (BASIC-T), we denote by Σ^ι and Σ^o the signatures extracted from ι and o, respectively; analogously, we denote by Γ^ι, $\Gamma^{\overline{\mu}}$ and $\Gamma^{\overline{\varphi}}$ the internal type environments extracted from ι, $\overline{\mu}$ and $\overline{\varphi}$, respectively, formally:

$$(\textsc{structural-sub})\frac{}{\Delta \vdash C \leqslant C'\ \text{OK}}\ mtype(\Delta, C', N) = T' \Rightarrow mtype(\Delta, C, N) = T, \Delta \vdash T \leqslant T'$$

$$(\textsc{method-sub})\frac{\Delta \vdash C'_i \leqslant C_i \quad \forall i \in 1..n \qquad \Delta \vdash C \leqslant C'}{\Delta \vdash C_1 \dots C_n \to C \leqslant C'_1 \dots C'_n \to C'} \qquad (\textsc{refl-sub})\frac{}{\Delta \vdash C \leqslant C}\ \Delta(C) = \_$$

$$(\textsc{decl-sub})\frac{}{\Delta \vdash C \leqslant C_i}\ \begin{array}{l}\Delta(C) = [\_; \_; \_; C_1 \dots C_n] \\ i \in 1..n\end{array} \qquad (\textsc{trans-sub})\frac{\Delta \vdash C_1 \leqslant C_2 \quad \Delta \vdash C_2 \leqslant C_3}{\Delta \vdash C_1 \leqslant C_3}$$

Fig. 9. Subtyping relation.

$$(\textsc{var-t})\frac{}{\Delta; \Gamma; I; \Pi \vdash x : C}\ \Pi(x) = C \qquad (\texttt{this-t})\frac{}{\Delta; \Gamma; C_1 \dots C_n; \Pi \vdash \texttt{this} : C_i}\ i \in 1..n$$

$$(\textsc{client-field-t})\frac{\Delta; \Gamma; I; \Pi \vdash e_0 : C_0}{\Delta; \Gamma; I; \Pi \vdash e_0.F : C}\ mtype(\Delta, C_0, F) = C$$

$$(\textsc{client-invk-t})\frac{\Delta; \Gamma; I; \Pi \vdash e_0 : C_0 \qquad \Delta; \Gamma; I; \Pi \vdash e_i : C'_i\ \forall i \in 1..n}{\Delta; \Gamma; I; \Pi \vdash e_0.M(e_1 \dots e_n) : C}\ \begin{array}{l}mtype(\Delta, C_0, M) = C_1 \dots C_n \to C \\ \Delta \vdash C'_i \leqslant C_i\ \forall i \in 1..n\end{array}$$

$$(\textsc{int-field-t})\frac{}{\Delta; \Gamma; I; \Pi \vdash f : C}\ \Gamma(f) = C$$

$$(\textsc{int-invk-t})\frac{\Delta; \Gamma; I; \Pi \vdash e_i : C'_i\ \forall i \in 1..n}{\Delta; \Gamma; I; \Pi \vdash m(e_1 \dots e_n) : C}\ \begin{array}{l}\Gamma(m) = C_1 \dots C_n \to C \\ \Delta \vdash C'_i \leqslant C_i\ \forall i \in 1..n\end{array}$$

$$(\textsc{new-t})\frac{\Delta; \Gamma; I; \Pi \vdash e_i : C'_i\ \forall i \in 1..n}{\Delta; \Gamma; I; \Pi \vdash \texttt{new}\ C(e_1 \dots e_n) : C}\ \begin{array}{l}\Delta(C) = [\emptyset; \_; C_1 \dots C_n; \_] \\ \Delta \vdash C'_i \leqslant C_i\ \forall i \in 1..n\end{array}$$

Fig. 10. Typing rules for expressions.

$$\begin{array}{lll}
\Sigma^o = N_1 : T_1 \dots N_k : T_k & \text{with } o = N_1 : T_1 \mapsto n_1 \dots N_k : T_k \mapsto n_k \\
\Sigma^\iota = N_1 : T_1 \dots N_k : T_k & \text{with } \iota = n_1 : T_1 \mapsto N_1 \dots n_k : T_k \mapsto N_k \\
\Gamma^\iota = n_1 : T_1 \dots n_k : T_k & \text{with } \iota = n_1 : T_1 \mapsto N_1 \dots n_k : T_k \mapsto N_k \\
\Gamma^{\overline{\mu}} = m_1 : \overline{C}\,\overline{x}_1 \to C_1 \dots m_k : \overline{C}\,\overline{x}_k \to C_k & \text{with } \overline{\mu} = C_1\ m_1(\overline{C}\,\overline{x}_1)\{\texttt{return}\ e_1;\} \\
 & \qquad \dots \\
 & \qquad C_k\ m_k(\overline{C}\,\overline{x}_k)\{\texttt{return}\ e_k;\} \\
\Gamma^{\overline{\varphi}} = f_1 : C_1 \dots f_k : C_k & \text{with } \overline{\varphi} = C_1\ f_1 \dots C_k\ f_k
\end{array}$$

A basic class is well typed w.r.t. Δ under three conditions. First, methods have their declared types w.r.t. Δ, the internal type environment assigning to member internal names their annotations, and the type in the supertypes (assumed as types for `this`). Second, the constructor has its declared type w.r.t. Δ and the internal type environment assigning to internal field names their annotations. Finally, type annotations in input signature, output signature and local part must be consistent, that is, a virtual member can be used inside the class with a supertype of its exported type (first side condition), and a member can be exported with a supertype of its internal type (second side condition).

Typing rules for sum, reduct and freeze are based on those in [3]. Rule (SUM-T) imposes the same constructor type and disjoint output signatures. In (REDUCT-T), the judgment $\Delta \vdash \sigma : \Sigma \to \Sigma'$ means that, if σ maps $N : T$ into $N' : T'$, then $\Delta \vdash T' \leqslant T$ holds. Hence, the side condition allows a member to be imported with a more specific type, and exported with a more general type. Analogously, rule (THIS-TYPE-T) allows the type of `this` to become more specific.

Typing rules in Fig. 10 define the judgment $\Delta; \Gamma; I; \Pi \vdash e : C$ for well-typed expressions.

They are analogous to FJ rules. However, note that member type is found in receiver's class for client member selection, whereas it is found in the internal type environment for internal member selection. Also, note that (NEW-T) requires a class to have an empty input signature in order to be instantiated (see comment to rule (OBJ-CREATION) in previous section).

Finally, typing rules in Fig. 11 define the judgment $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e : C$ for well-typed runtime expressions. These expressions are typed using an additional type environment Δ^r, which gives for each instantiable class the types of its internal field names.

$$(\textsc{block-t})\frac{\begin{array}{l}\Delta;\Delta^r;\Gamma;I;\Pi\vdash v:C'\\ \Delta;\Delta^r;\Gamma';C';\vdash\overline{\mu}:\Gamma^{\overline{\mu}}\\ \Delta;\Delta^r;\Gamma';I;\Pi\vdash e:C\end{array}}{\Delta;\Delta^r;\Gamma;I;\Pi\vdash[\overline{\mu};v\mid e]:C}\ \Gamma'=\Gamma,\Delta^r(C'),\Gamma^{\overline{\mu}}$$

$$(\textsc{pre-obj-t})\frac{\Delta;\Delta^r;\Gamma;I;\Pi\vdash e_i:C_i'\ \forall i\in 1..n}{\Delta;\Delta^r;\Gamma;I;\Pi\vdash C(f_1=e_1;\dots f_n=e_n;):C}\ \begin{array}{l}\Delta^r(C)=f_1{:}C_1\dots f_n:C_n\\ \Delta\vdash C_i'\leqslant C_i\ \forall i\in 1..n\end{array}$$

Fig. 11. Typing rules for runtime expressions.

Rule (BLOCK-T) checks that the current object is well typed and, moreover, that the enclosed method declarations and expression are well typed in the internal type environment corresponding to the current object's class in Δ^r. In this case, the type of the block is that of the enclosed expression. Rule (PRE-OBJ-T) checks that each initialization expressions has a subtype of the type of the corresponding field internal name, found in the internal type environment associated to the (pre)object's class in Δ^r. Rules for other forms of expressions are analogous to those in Fig. 10, plus propagation of the runtime class type environment.

Soundness of the type system is expressed by Theorem 1, Theorem 9 and Theorem 13 in the following. The first states that flattening of a well-typed program always terminates and preserves its type, the others state that reduction of a well-typed ground expression in the context of a well-typed flat program does not get stuck, since progress and subject reduction properties hold. In particular, progress (Theorem 9) is proved as a corollary of an *extended progress* property (Theorem 8).

Theorem 1 *(Soundness w.r.t. flattening relation). If* $\vdash p:\Delta$, *then* $p \xrightarrow{\star} p'$ *for some* p' *flat program, and* $\vdash p':\Delta$.

Proof. The proof is a simple adaptation of that given in [3]. □

In order to prove extended progress, we need the following lemmas.

For p flat program, let us write $\vdash p:\langle\Delta,\Delta^r\rangle$ if $\vdash p:\Delta$ and Δ^r is the runtime class type environment extracted from p. That is, for each basic class declaration of form $C\mapsto[\emptyset\mid o\mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}]$ in p, $\Delta^r_p(C)=\Gamma^{\overline{\varphi}}$.

Lemma 2. *If* $\Delta;\Delta^r;\Gamma;I;\Pi\vdash\mathcal{E}\{e\}:C$ *and* $HB(\mathcal{E})=\emptyset$, *then, for some* C', $\Delta;\Delta^r;\Gamma;I;\Pi\vdash e:C'$.

Proof. By structural induction on $\mathcal{E}$. □

Here and in what follows we use the notation # for the length of a sequence.

Lemma 3. *If* $\vdash p:\Delta$, *then*:

(i) $dom(p)=dom(\Delta)$;
(ii) *If* $\Delta(C)=[\Sigma^\iota;\Sigma^o;\overline{C};C]$ *and* $p(C)=[\iota\mid o\mid I\ \{\overline{\varphi}\ (\overline{C\,x})\{\overline{f=e}\}\ \overline{\mu}\}]$; *then*
  A: $img(\iota)=dom(\Sigma^\iota)$,
  B: $dom(o)=dom(\Sigma^o)$,
  C: $\#(\overline{C\,x})=\#(\overline{C})$,
  D: $\Delta\vdash p(C):\Delta(C)$,
  E: $\Delta;\Gamma^\iota,\Gamma^{\overline{\mu}},\Gamma^{\overline{\varphi}};C\vdash\overline{\mu}:\Gamma^{\overline{\mu}}$.

Proof. Since the judgment has been derived by rules (PROG-T) and (BASE-T). □

Lemma 4. *If* $\Delta;\Delta^r;\Gamma;I;\Pi\vdash v^C:C$, *then* $\Delta(C)=[\emptyset\mid o\mid\rho]$.

Proof. By rule (PRE-OBJ-T) and definition of Δ^r. □

Lemma 5. *If* $\vdash p:\Delta$, $\Delta(C)=[\emptyset\mid o\mid\rho]$ *and* $mtype(\Delta,C,N)=T$, *then*:

(i) $N\mapsto T$ *is contained in the output map of* $\Delta(C)$.
(ii) $N:T\mapsto n$ *is contained in the output map of* $p(C)$.

Proof. (i) By definition of $mtype(\Delta,C,N)=T$ we obtain $N\mapsto T\in o$.
(ii) By (i) and Lemma 3(i), Lemma 3(ii, A, B, D). □

Lemma 6. *If* $\Delta; \Delta^r; \Gamma; \Pi; [\overline{\mu}; v^{C'} \mid \mathcal{E}\{m(\bar{e})\}] \vdash C :$ *and* $HB(\mathcal{E}) = \emptyset$, *then* $\overline{\mu}(m) = \langle \bar{x}, e \rangle$ *and* $\#(\bar{e}) = \#(\bar{x})$.

Proof. By well-formedness of the block we have $\overline{\mu}(m) = \langle x_1 \dots x_n, e \rangle$. The judgment has been derived by (BLOCK-T), hence $\Delta; \Delta^r; \Gamma, \Delta^r(C'), \Gamma^{\overline{\mu}}; I; \Pi \vdash \mathcal{E}\{m(\bar{e})\} : C$. Then by Lemma 2 we have $\Delta; \Delta^r; \Gamma, \Delta^r(C'), \Gamma^{\overline{\mu}}; \Pi; m(\bar{e}) \vdash C'' :$. This judgment has been derived by rule (INT-INVK-T), that requires $\#(\bar{e}) = n$. □

Lemma 7. *If* v^C *is well typed and* $p(C) = [\iota \mid o \mid I \ \{\overline{\varphi}\ (\overline{C\,x})\{f_1 = e_1 \dots f_n = e_n\}\ \overline{\mu}\}]$, *then* $dom(\overline{\varphi}) = dom(v^C) = \{f_1, \dots, f_n\}$.

Proof. By rule (BASIC-T) and rule (K-T) we have $dom(\overline{\varphi}) = \{f_1, \dots, f_n\}$. By rule (PRE-OBJ-T) and definition of Δ^r we have $dom(v^C) = dom(\overline{\varphi})$. □

Theorem 8 *(Extended progress). If* $\vdash p : \langle \Delta, \Delta^r \rangle$ *and* $\Delta; \Delta^r; \Gamma; \emptyset; e \vdash C :$ *then one of the following cases holds*:

(1) *e is a value.*
(2) *e is of the form* $\mathcal{E}\{f\}$ *with* $f \notin HB(\mathcal{E})$ *and* $f \in dom(\Gamma)$.
(3) *e is of the form* $\mathcal{E}\{m(\overline{v})\}$ *with* $m \notin HB(\mathcal{E})$ *and* $m \in dom(\Gamma)$.
(4) $e \longrightarrow_p e'$ *for some* e'.

Proof. By induction on the typing rules.

(VAR-T) This case is empty since $\Pi = \emptyset$.

(CLIENT-FIELD-T) The term is of the form $e_0.F$ and we have:
A: $\Delta; \Delta^r; \Gamma; \emptyset; e_0.F \vdash C :$,
B: $\Delta; \Delta^r; \Gamma; \emptyset; e_0 \vdash C_0 :$,
C: $mtype(\Delta, C_0, F) = C$.
From (B) by inductive hypothesis e_0 verifies one of cases (1)–(4).
- If e_0 verifies case (1), then the term is of the form $v^{C_0}.F$. We can apply rule (CLIENT-FIELD) since the implicit and explicit side conditions are verified:
  - $p(C_0) = [\iota \mid o, F : C \mapsto f \mid I \ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}]$ by (B), (C), Lemma 4 and Lemma 5(ii); $f \in \overline{\varphi}$ holds by well-formedness of class C_0,
  - $[\overline{\mu}; v^{C_0} \mid f]$ is well formed since $f \in dom(v^{C_0})$ by Lemma 7.
- If e_0 verifies case (2) or (3), then the term verifies case (2) or (3), respectively, as well.
- If e_0 verifies case (4), then the term reduces by $(\mathcal{E})$.

(CLIENT-INVK-T) The term is of the form $e_0.M(e_1 \dots e_n)$, and we have:
A: $\Delta; \Delta^r; \Gamma; \emptyset; e_0.M(e_1 \dots e_n) \vdash C :$,
B: $\Delta; \Delta^r; \Gamma; \emptyset; e_0 \vdash C_0 :$, $\Delta; \Delta^r; \Gamma; \emptyset; e_i \vdash C_i' :$ for all $i \in 1..n$,
C: $mtype(\Delta, C_0, M) = C_1 \dots C_n \to C$.
From (B) by inductive hypothesis $e_0, e_1, \dots, e_n$ verify one of cases (1)–(4).
If all verify case (1), then the term is of the form $v^{C_0}.M(\overline{v})$. We can apply (CLIENT-INVK) since the implicit and explicit side conditions are verified:
- $p(C) = [\iota \mid o, M : C_1 \dots C_n \to C \mapsto m \mid I \ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}]$ by (B), (C) and Lemma 5(ii),
- $[\overline{\mu}; v^{C_0} \mid m(\overline{v})]$ is well formed since $m \in \overline{\mu}$ by well-formedness of class C_0.

If there exists one e_i that verifies case (2) or (3), then the term verifies case (2) or (3), respectively, as well. If there exists one e_i that verifies case (4), then the term reduces by $(\mathcal{E})$.

(INT-FIELD-T) The term verifies case (2).

(INT-INVK-T) The term is of the form $m(e_1 \dots e_n)$ and we have:
A: $\Delta; \Delta^r; \Gamma; \emptyset; m(e_1 \dots e_n) \vdash C :$,
B: $\Delta; \Delta^r; \Gamma; \emptyset; e_0 \vdash C_0 :$, $\Delta; \Delta^r; \Gamma; \emptyset; e_i \vdash C_i' :$ for all $i \in 1..n$,
From (B) by inductive hypothesis $e_0, e_1, \dots, e_n$ verify one of cases (1)–(4).
If all verify case (1), then the term is of the form $m(\overline{v})$, hence verifies case (3). If there exists one e_i that verifies case (2) or (3), then the term verifies case (2) or (3), respectively, as well. If there exists one e_i that verifies case (4), then the term reduces by $(\mathcal{E})$.

(NEW-T) The term is of the form $\texttt{new}\ C(e_1 \dots e_n)$ and we have:
A: $\Delta; \Delta^r; \Gamma; \emptyset; \texttt{new}\ C(e_1 \dots e_n) \vdash C :$,
B: $\Delta; \Delta^r; \Gamma; \emptyset; e_i \vdash C_i' :$ for all $i \in 1..n$,
C: $\Delta(C) = [\emptyset; \Sigma^o; C_1 \dots C_n; C']$.
From (B) by inductive hypothesis $e_1, \dots, e_n$ verify one of cases (1)–(4).
If all verify case (1), then the term is of the form $\texttt{new}\ C(\overline{v})$. We can apply (OBJ-CREATION) since the implicit and explicit side conditions are verified:
- $p(C) = [\emptyset \mid o \mid I \ \{\overline{\varphi}\ (C_1 x_1, \dots, C_n x_n)\{\overline{f = e}\}\ \overline{\mu}\}]$ follows from (B) by Lemma 3(i), Lemma 3(ii, A), and Lemma 3(ii, C).

If there exists one e_i that verifies case (2) or (3), then the term verifies case (2) or (3), respectively, as well. If there exists one e_i that verifies case (4), then the term reduces by $(\mathcal{E})$.

(BLOCK-T) The term is of the form $[\overline{\mu}; C(f_1 = v_1 \dots f_n = v_n) \mid e]$ and we have:
A: $\Delta; \Delta^r; \Gamma; \emptyset; \emptyset \vdash [\overline{\mu}; C(f_1 = v_1 \dots f_n = v_n) \mid e] : C$,
B: $\Delta; \Delta^r; \Gamma'; \emptyset; \emptyset \vdash e : C$.
From (B) by inductive hypothesis e verifies one of cases (1)–(4).
- If e verifies case (1), then the term reduces by (EXIT-BLOCK).
- If e verifies case (2), that is, e is of the form $\mathcal{E}\{f\}$ with $f \notin HB(\mathcal{E})$ and $f \in dom(\Gamma)$, then we can apply (INT-FIELD), since the implicit and explicit side conditions are verified:
  - $f \notin HB(\mathcal{E})$,
  - $f = f_i$ by well-formedness of the block.
- If e verifies case (3), that is e is of the form $\mathcal{E}\{m(\overline{v})\}$ with $m \notin HB(\mathcal{E})$ and $m \in dom(\Gamma)$, then we can apply (INT-INVK), since the implicit and explicit side conditions are verified:
  - $m \notin HB(\mathcal{E})$,
  - $\#(\overline{v}) = \#(\overline{x})$ by Lemma 6.
- If e verifies cases (4), then the term reduces by $(\mathcal{E})$.

(PRE-OBJ-T) The term is of the form $C(f_1 = e_1; \dots f_n = e_n;)$ and we have:
A: $\Delta; \Delta^r; \Gamma; \emptyset; \emptyset \vdash C(f_1 = e_1 \dots f_n = e_n) : C$,
B: $\Delta; \Delta^r; \Gamma; \emptyset; \emptyset \vdash e_i : C'_i$ for all $i \in 1..n$.
From (B) by inductive hypothesis $e_1, \dots, e_n$ verify one of cases (1)–(4).

If all verify case (1), then the term is of the form $C(f_1 = v_1 \dots f_n = v_n)$, hence the term verifies case (1) as well. If there exists one e_i that verifies case (2) or (3), then the term verifies case (2) or (3), respectively, as well. If there exists one e_i that verifies case (4), then the term reduces by $(\mathcal{E})$. □

Theorem 9 *(Progress). If* $\vdash p : \Delta$ *and* $\Delta; \Delta^r_p; \emptyset; \emptyset; \emptyset \vdash e : C$, *then either* e *is a value or* $e \longrightarrow_p e'$ *for some* e'.

Proof. The thesis follows from extended progress (Theorem 8). Indeed case (2) and (3) cannot occour since $\Gamma = \emptyset$. □

The following lemmas are needed to prove the subject reduction property (Theorem 13).

Lemma 10 *(Weakening). If* $\Delta; \Delta^r; \emptyset; \emptyset; \emptyset \vdash e : C$, *then* $\Delta; \Delta^r; \Gamma; \Pi; e \vdash C$ *: for any* Γ, Π.

Proof. By induction on the derivation of $\Delta; \Delta^r; \emptyset; \emptyset; \emptyset \vdash e : C$. □

Lemma 11. *If* $HB(\mathcal{E}) = \emptyset$, $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \mathcal{E}\{e_1\} : C$, $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e_1 : C_1$, $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e_2 : C_2$ *and* $\Delta \vdash C_2 \leqslant C_1$, *then* $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \mathcal{E}\{e_2\} : C'$ *and* $\Delta \vdash C' \leqslant C$.

Proof. By induction on the derivation of $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \mathcal{E}\{e_1\} : C$. □

Lemma 12 *(Substitution). If* $\Delta; \Delta^r; \Gamma; I; \Pi, x_1 : C'_1 \dots x_n : C'_n \vdash e : C$ *and, for all* $i \in 1..n$, $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e_i : C_i$ *and* $\Delta \vdash C_i \leqslant C'_i$, *then* $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e[e_1/x_1, \dots e_n/x_n] : C'$ *and* $\Delta \vdash C' \leqslant C$.

Proof. By induction on the derivation of $\Delta; \Delta^r; \Gamma; I; \Pi, x_1 : C'_1 \dots x_n : C'_n \vdash e : C$. □

Theorem 13 *(Subject reduction). If* $\vdash p : \langle \Delta, \Delta^r \rangle$, $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e : C$ *and* $e \longrightarrow_p e'$, *then* $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e' : C'$ *and* $\Delta \vdash C' \leqslant C$.

Proof. By induction on the reduction rules.

$(\mathcal{E})$ Straightforward structural induction on $\mathcal{E}$.

(CLIENT-FIELD) The term is of the form $v^C.F$ and we have:
A: $v^C.F \longrightarrow_p [\overline{\mu}; v^C \mid f]$,
B: $p(C) = [\iota \mid o \mid \rho]$,
C: $o(F) = f$.
Moreover $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v^C.F : C'$ holds by (CLIENT-FIELD-T), hence we have:
D: $\Delta; \Delta^r; \Gamma; \mathit{impl}; \Pi \vdash v^C : C$,
E: $\mathit{mtype}(\Delta, C, F) = C'$.
Note that $F : C' \mapsto f \in o$ by the definition of $\mathit{mtype}(\Delta, C, F)$ and (C). Set $\Gamma' \equiv \Gamma, \Gamma^{\overline{\mu}}, \Delta^r(C)$; then $\Delta; \Delta^r; \Gamma; I; \Pi \vdash [\overline{\mu}; v^C \mid f] : C''$ holds by (BLOCK-T) since all the premises hold:

- $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v^C : C$ by (D);
- $\Delta; \Gamma'; C \vdash \overline{\mu} : \Gamma^{\overline{\mu}}$ by Lemma 3(ii, E);
- $\Delta; \Delta^r; \Gamma'; \emptyset; \emptyset \vdash f : C''$ by (INT-FIELD-T) since the side condition $\Gamma'(f) = C''$ is verified; indeed, $C''\ f \in \overline{\varphi}$ by well-formedness of $p(C)$ and definition of $\Delta^r(C)$.

Since $\vdash p : \langle \Delta, \Delta^r \rangle$ holds, by Lemma 3(ii, D) (BASE-T) holds, hence we get $\Delta \vdash C'' \leqslant C'$, in fact $F : C' \mapsto f \in o$ and $C''\ f \in \overline{\varphi}$. (BASE-T), $F : C' \mapsto f \in o$ and $C''\ f \in \overline{\varphi}$.

(CLIENT-INVK) The term is of the form $v^{C_0}.M(v_1, \ldots, v_n)$ and we have:

A: $v^{C_0}.M(v_1, \ldots, v_n) \longrightarrow_p [\overline{\mu}; v^{C_0} \mid m(v_1 \ldots v_n)]$,
B: $p(C_0) = [\iota \mid o \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}]$,
C: $o(M) = m$.

Moreover $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v^{C_0}.M(v_1 \ldots v_n) : C$ holds by (CLIENT-INVK-T). Hence we have:

D: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v^{C_0} : C_0$,
F: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v_i : C_i''$ for $i \in 1..n$,
G: $mtype(\Delta, C_0, M) = C_1 \ldots C_n \to C$,
H: $\Delta \vdash C_i'' \leqslant C_i$ for $i \in 1..n$.

Note that $M : C_1 \ldots C_n \mapsto C \mapsto m \in o$ by the definition of $mtype(\Delta, C, M)$ and (C). Set $\Gamma' \equiv \Gamma, \Delta^r(C_0), \Gamma^{\overline{\mu}}$; then $\Delta; \Delta^r; \Gamma; I; \Pi \vdash [\overline{\mu}; v^{C_0} \mid m(v_1 \ldots v_n)] : C$ holds by (BLOCK-T) since all the premises hold:

- $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v^{C_0} : C_0$ by (D);
- $\Delta; \Gamma'; C \vdash \overline{\mu} : \Gamma^{\overline{\mu}}$ by Lemma 3(ii, E);
- $\Delta; \Delta^r; \Delta^r(C_0); \emptyset; \emptyset \vdash m(v_1 \ldots v_n) : C'$ holds by (INT-INVK-T), since the side conditions and the premises hold:
  - $\Gamma'(m) = C_1' \ldots C_n' \to C'$ is verified since $C'm(C_1'\ x_1 \ldots C_n'\ x_n)\{\texttt{return}\ \_;\} \in \overline{\mu}$ by well-formedness of $p(C)$ and definition of $\Gamma^{\overline{\mu}}$.
  - $\Delta \vdash C_i' \leqslant C_i$ for $i \in 1..n$ holds by (H) and (TRANS-S) since $\Delta \vdash C_i' \leqslant C_i''$ for $i \in 1..n$ holds by (BASE-T). ((BASE-T) holds by Lemma 3(ii, D) since $\vdash p : \langle \Delta, \Delta^r \rangle$ holds.)
  - $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v_i : C_i''$ for $i \in 1..n$ holds by (F) and (TRANS-S).

Since $\vdash p : \langle \Delta, \Delta^r \rangle$ holds, by Lemma 3(ii, D) (BASE-T) holds, hence we get $\Delta \vdash C' \leqslant C$, in fact $M : C_1 \ldots C_n \to C \mapsto m \in o$ and $C'm(C_1'\ x_1 \ldots C_n'\ x_n)\{\texttt{return}\ \_;\} \in \overline{\mu}$.

(INT-FIELD) The term is of the form $[\overline{\mu}; C(f_1 = v_1 \ldots f_n = v_n) \mid \mathcal{E}\{f_i\}]$ and we have:

A: $[\overline{\mu}; C(f_1 = v_1 \ldots f_n = v_n) \mid \mathcal{E}\{f_i\}] \longrightarrow_p [\overline{\mu}; C(f_1 = v_1 \ldots f_n = v_n) \mid \mathcal{E}\{v_i\}]$,
B: $HB(\mathcal{E}) = \emptyset$.

Set $\Gamma' \equiv \Gamma, \Gamma^{\overline{\mu}}, \Delta^r(C)$; then $\Delta; \Delta^r; \Gamma; I; \Pi \vdash [\overline{\mu}; C(f_1 = v_1 \ldots f_n = v_n) \mid \mathcal{E}\{f_i\}] : C'$ holds by (BLOCK-T), hence we have:

C: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash C(f_1 = v_1 \ldots f_n = v_n) : C$,
D: $\Delta; \Gamma'; C \vdash \overline{\mu} : \Gamma^{\overline{\mu}}$,
E: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \mathcal{E}\{f_i\} : C'$.

Moreover from (E) we know that

F: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash f_i : C_i$ by (INT-FIELD-T).

The judgment $\Delta; \Delta^r; \Gamma; I; \Pi \vdash [\overline{\mu}; C(f_1 = v_1 \ldots f_n = v_n) \mid \mathcal{E}\{v_i\}] : C''$ holds by rule (BLOCK-T), since all the premises hold:

- $\Delta; \Delta^r; \Gamma; I; \Pi \vdash C(f_1 = v_1 \ldots f_n = v_n) : C$ by (C);
- $\Delta; \Gamma'; C \vdash \overline{\mu} : \Gamma^{\overline{\mu}}$ by (D);
- $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \mathcal{E}\{v_i\} : C''$ holds by Lemma 11 by (A), (B), (F) and since
  - $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v_i : C_i'$ holds by (C) and (PRE-OBJ-T).
  - $\Delta \vdash C_i \leqslant C_i'$ holds by(C), (PRE-OBJ-T) and (F).

  Indeed $\Delta \vdash C'' \leqslant C'$ by Lemma 11.

(INT-INVK) The term is of the form $[\overline{\mu}; v^{C_0} \mid \mathcal{E}\{m(v_1 \ldots v_n)\}]$ and we have:

A: $[\overline{\mu}; v^{C_0} \mid \mathcal{E}\{m(v_1 \ldots v_n)\}] \longrightarrow_p [\overline{\mu}; v^{C_0} \mid \mathcal{E}\{e[v_1/x_1 \ldots v_n/x_n][v^{C_0}/\texttt{this}]\}]$,
B: $HB(\mathcal{E}) = \emptyset$,
C: $\overline{\mu}(m) = \langle x_1 \ldots x_n, e \rangle$.

Set $\Gamma' \equiv \Gamma, \Gamma^{\overline{\mu}}, \Delta^r(C_0)$; then $\Delta; \Delta^r; \Gamma; I; \Pi \vdash [\overline{\mu}; v^{C_0} \mid \mathcal{E}\{m(v_1 \ldots v_n)\}] : C$ holds by (BLOCK-T), hence we have:

C: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v^{C_0} : C_0$,
D: $\Delta; \Gamma'; C_0 \vdash \overline{\mu} : \Gamma^{\overline{\mu}}$,
E: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \mathcal{E}\{m(v_1 \ldots v_n)\} : C$.

Moreover from (E) we know that

F: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash m(v_1 \ldots v_n) : C'$ by (INT-INVK-T).

This implies also

G: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v_i : C_i$ for all $i \in 1..n$,
H: $\Gamma(m) = C_1' \ldots C_n' \to C'$,
I: $\Delta \vdash C_i \leqslant C_i'$ for all $i \in 1..n$.

The judgment $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e[v_1/x_1 \ldots v_n/x_n][v^{C_0}/\texttt{this}] : C''$ holds by (METHOD-T), (METHODS-T) and (BASIC-T), (G), (H), (I), and Lemma 12.

Moreover $\Delta \vdash C'' \leqslant C'$ holds by (C) and Lemma 12.

We conclude that the judgment

$\Delta; \Delta^r; \Gamma; I; \Pi \vdash [\overline{\mu}; v^{C_0} \mid \mathcal{E}\{e[v_1/x_1 \dots v_n/x_n][v^{C_0}/\texttt{this}]\}] : C'''$ holds by rule (BLOCK-T), since all the premises hold:

- $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v^{C_0} : C_0$ by (C);
- $\Delta; \Gamma'; C_0 \vdash \overline{\mu} : \Gamma^{\overline{\mu}}$ by (D);
- $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \mathcal{E}\{e[v_1/x_1 \dots v_n/x_n][v^{C_0}/\texttt{this}]\} : C'''$ holds by Lemma 11,
  $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e[v_1/x_1 \dots v_n/x_n][v^{C_0}/\texttt{this}] : C''$, and $\Delta \vdash C'' \leqslant C'$. Indeed $\Delta \vdash C''' \leqslant C$ by Lemma 12.

(OBJ-CREATION) The term is of the form $\texttt{new}\ C(v_1 \dots v_n)$ and we have:

A: $\texttt{new}\ C(v_1 \dots v_n) \longrightarrow_p C(\phi[\overline{v}/\overline{x}])$,
B: $p(C)=[\emptyset \mid o \mid \rho]$,
C: $\rho=I\ \{\overline{\varphi}\ (C_1\ x_1 \dots C_n\ x_n)\{\phi\}\ \overline{\mu}\}$.

Moreover $\Delta; \Delta^r; \Gamma; I; \Pi \vdash \texttt{new}\ C(v_1 \dots v_n) : C$ holds by (NEW-T). Hence we have:

D: $\Delta; \Delta^r; \Gamma; I; \Pi \vdash v_i : C_i$ for all $i \in 1..n$,
E: $\Delta(C) = [\emptyset; \_; C'_1 \dots C'_n; \_]$,
F: $\Delta \vdash C_i \leqslant C'_i$ for all $i \in 1..n$.

Set $\phi \equiv f_1 = e_1, \dots f_k = e_k$, by (BASIC-T), (K-T) and definition of $\Gamma^{\overline{\varphi}}$ we have:

G: $\overline{\varphi} = C'''_1\ f_1 \dots C'''_k\ f_k$,
H: $\Delta; \emptyset; x_1 : C_1 \dots x_n : C_n; e_i \vdash C''_i :$ for all $i \in 1..k$,
I: $\Delta \vdash C''_i \leqslant C'''_i$ for all $i \in 1..k$.

For all $i \in 1..k$, the judgments

L: $\Delta; \emptyset; \emptyset; e_i[v_1/x_1 \dots v_n/x_n] \vdash C^{IV}_i :$, and
M: $\Delta \vdash C^{IV}_i \leqslant C''_i$

hold by Lemma 12, (H), (D), and (F).

The judgment $\Delta; \Delta^r; \Gamma; I; \Pi \vdash C(\phi[v_1/x_1 \dots v_n/x_n]) : C$ holds by (PRE-OBJ-T) since all the premises hold:

- $\Delta; \Delta^r; \Gamma; I; \Pi \vdash e_i[v_1/x_1 \dots v_n/x_n] : C^{IV}_i$ for all $i \in 1..n$ by (L) and Lemma 10,
- $\Delta \vdash C^{IV}_i \leqslant C'''_i$ by (M), (I) and (S-TRANS).

(EXIT-BLOCK) Trivial. □

4. Direct semantics

Direct semantics allows a modular approach where each class (module) can be analyzed (notably, compiled) in isolation, since references to other classes do not need to be resolved before runtime. In this case, look-up is a non trivial procedure where a class member (e.g., method) is retrieved from other classes and possibly modified as effect of the module operators. Since FJIG subsumes a variety of mechanisms for class composition, including standard inheritance, mixins, traits, and hiding, the definition of direct semantics for FJIG provides a guideline which can be emulated by real extensions of class-based languages, and also a hint to implementation.

Before giving the formal definition, we illustrate how direct semantics works by some examples. To begin, let us consider a program[10] where a class is defined as the sum of two others:

$$
\begin{aligned}
C \mapsto\ & C_1 + C_2 \\
C_1 \mapsto\ & [\emptyset \mid \dots \mid \{\texttt{int}\ f;\ ()\{f = 3\}\ \dots\}] \\
C_2 \mapsto\ & [\emptyset \mid \dots, M \mapsto m \mid \{ \\
& \quad \texttt{int}\ f; \\
& \quad ()\{f = 5\} \\
& \quad \texttt{int}\ m1()\{\texttt{return}\ m2 + 1;\} \\
& \quad \texttt{int}\ m2()\{\texttt{return}\ f + 1;\}\}]
\end{aligned}
$$

and take $\texttt{new}\ C().M()$ as expression to be reduced. The invocation of the constructor of C triggers *constructor lookup*. In this case, since C is the sum of C_1 and C_2, the result of the lookup is the constructor obtained by merging those of C_1 and C_2, concatenating their bodies. However, since both classes have a local field named f, initialized to 3 and 5, respectively, they should be α-renamed in the sum. In direct semantics, we assume a standard α-renaming which adds `.1` and `.2` to internal names inherited from the first and second argument of the sum, respectively. Hence, the above expression reduces to $C(f.1 \mapsto 3, f.2 \mapsto 5).M()$. Now, M is invoked, triggering *method lookup*. The search is propagated to both C_1 and C_2. Only the lookup in C_2 is successful and returns the result

$$[;;\ \texttt{int}\ m()\{\texttt{return}\ f + 1;\} \mid m()]$$

[10] In order to write more readable examples, we assume integer values and operations, and omit default constructor.

π	$::= i_1 \dots i_k$	path ($i \in \{1,2\}$)
$\hat{N}$	$::= N \mid \pi$	member reference (external name or path)
$\hat{\iota}$	$::= n_1 \mapsto \pi_1 \dots n_k \mapsto \pi_k$	path map
e	$::= \dots \mid [\hat{\iota}; \overline{\mu}; v \mid e]$	(generalized) block
λ	$::= [\iota; \hat{\iota}; \overline{\mu} \mid n]$	lookup result

$$(\mathcal{E})\ \frac{e \longrightarrow_p e'}{\mathcal{E}\{e\} \longrightarrow_p \mathcal{E}\{e'\}} \qquad (\text{CLIENT-FIELD})\ \frac{}{v^C.F \longrightarrow_p [\hat{\iota}; \overline{\mu}; v^C \mid f]}\ lookup_p\langle F, C\rangle = [\hat{\iota}; \overline{\mu} \mid f]$$

$$(\text{CLIENT-INVK})\ \frac{}{v^C.M(\overline{v}) \longrightarrow_p [\hat{\iota}; \overline{\mu}; v^C \mid m(\overline{v})]}\ lookup_p\langle M, C\rangle = [\hat{\iota}; \overline{\mu} \mid m]$$

$$(\text{INT-FIELD})\ \frac{}{[\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{f\}] \longrightarrow_p [\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{v_i\}]}\ \begin{array}{l} f \notin HB(\mathcal{E}) \\ v = C(f_1{=}v_1 \dots f_n{=}v_n) \\ f = f_i \end{array}$$

$$(\text{INT-INVK})\ \frac{}{[\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{m(\overline{v})\}] \longrightarrow_p [\hat{\iota}; \overline{\mu}; v \mid \mathcal{E}\{e[\overline{v}/\overline{x}][v^C/\texttt{this}]\}]}\ \begin{array}{l} m \notin HB(\mathcal{E}) \\ \overline{\mu}(m) = \langle \overline{x}, e\rangle \end{array}$$

$$(\text{PATH})\ \frac{}{[\hat{\iota}, n \mapsto \pi; \overline{\mu}; v^C \mid e] \longrightarrow_p [\hat{\iota}, \hat{\iota}'; \overline{\mu}[n'/n], \overline{\mu}'; v^C \mid e[n'/n]]}\ \begin{array}{l} n \in names(e) \\ lookup_p\langle \pi, C\rangle = [\hat{\iota}'; \overline{\mu}' \mid n'] \end{array}$$

$$(\text{OBJ-CREATION})\ \frac{}{\texttt{new}\ C(\overline{v}) \longrightarrow_p C(\overline{f = e[\overline{v}/\overline{x}]})}\ \begin{array}{l} k\text{-}lookup_p(C) = (C_1\ x_1 \dots C_n\ x_n)\{\phi\} \\ \overline{x} = x_1 \dots x_n \end{array}$$

$$(\text{EXIT-BLOCK})\ \frac{}{[\hat{\iota}; \overline{\mu}; v \mid e] \longrightarrow_p e}\ names(e) = \emptyset$$

Fig. 12. Direct semantics.

The intuitive meaning is that we have found a method which is modified in $[;;\texttt{int}\ m()\{\texttt{return}\ f.2+1;\} \mid m()]$ to take into account that the method has been found in the second argument. Hence, this method invocation reduces to$[;\texttt{int}\ m()\{\texttt{return}\ f.2+1;\}; C(f.1 \mapsto 3, f.2 \mapsto 5) \mid m()]$ where the body of m correctly refers to the second field.

In flattening semantics, C reduces to the following basic class:

$$[\emptyset \mid \dots, M \mapsto m \mid \{\texttt{int}\ f.1; \texttt{int}\ f.2; \kappa\ \texttt{int}\ m()\{\texttt{return}\ f.2+1;\} \dots\}]$$
$$\kappa = (\,)\{f.1 = 3, f.2 = 5\}$$

Note that here the clash between the two fields is resolved during flattening (hence before runtime), by α-renaming. We have chosen as α-renaming the same used in direct semantics as an help for the reader, but of course in this case any other arbitrary α-renaming would work as well.

For instance, consider a program including

$$C \mapsto {}_{M_1 \mapsto M_1'|}C'_{|M \mapsto M'}$$
$$C' \mapsto [m' \mapsto M_1 \mid M' \mapsto m \mid \{\dots\ \texttt{int}\ m()\{\texttt{return}\ m'();\}\}]$$

and assume that some method invocation triggers the lookup for M in C. Then, the lookup is propagated under the name M' to C'. The lookup of M' in C' is successful and returns the result $[m' \mapsto M_1;;\texttt{int}\ m()\{\texttt{return}\ m'();\} \mid m()]$ which is modified in $[m' \mapsto M_1';;\texttt{int}\ m()\{\texttt{return}\ m'();\} \mid m()]$ as an effect of the input renaming.

In flattening semantics, C reduces to the following basic class:

$$[m' \mapsto M_1' \mid M \mapsto m \mid \{\dots\ \texttt{int}\ m()\{\texttt{return}\ m'();\}\}]$$

In order to give this definition, block expressions are generalized as shown in the top section of Fig. 12. First of all, we use *paths* to denote positions, that is, subterms, in a class expression. That is, 1 denotes the first direct subterm, and 2 denotes the second direct subterm (recall that class expressions are constructed by only unary and binary operators). An implementation could use a pointer to a node of the abstract syntax tree instead. We use the metavariable NT to range over external names or paths. Besides the previous components, a block contains a *path map* $\hat{\iota}$ which maps internal names to paths, which denote a subterm in the class expression defining the class C of the current object. More precisely, a path π always denotes a subterm of the form $freeze_N CE$, and is used as a permanent reference to the definition of member N in CE. Indeed, the external name N can be changed or removed by effect of outer reduct operators; however, references via π are not affected. Hence, when a reference π is encountered during current method execution, lookup of N in CE is

$$
\begin{array}{l}
\mathit{lookup}_p\langle \hat{N}, C\rangle = \mathit{lookup}_p\langle \hat{N}, \Lambda, C\rangle \\
\mathit{lookup}_p\langle \hat{N}, \pi, C\rangle = \mathit{lookup}_p\langle \hat{N}, \pi, CE\rangle \\
\qquad \text{if } p(C) = CE \\
\mathit{lookup}_p\langle N, \pi, [\iota \mid o, N \mapsto n \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}]\rangle = [\iota; \emptyset; \overline{\mu} \mid n] \\
\mathit{lookup}_p\langle \hat{N}, \pi, CE_1 + CE_2\rangle = \alpha_i([\iota; \hat{\iota}; \overline{\mu} \mid n]) \\
\qquad \text{if } \mathit{lookup}_p\langle \hat{N}, \pi.i, CE_i\rangle = [\iota; \hat{\iota}; \overline{\mu} \mid n],\ i \in \{1, 2\} \\
\mathit{lookup}_p\langle \hat{N}, \pi, {}_{\sigma^\iota|}CE_{|\sigma^o}\rangle = [\sigma^\iota \circ \iota; \hat{\iota}; \overline{\mu} \mid n] \\
\qquad \text{if } \mathit{lookup}_p\langle \hat{N}', \pi.1, CE\rangle = [\iota; \hat{\iota}; \overline{\mu} \mid n], \\
\qquad \hat{N}' = \sigma^o(N) \text{ if } \hat{N} = N, \hat{N}' = \hat{N} \text{ otherwise} \\
\mathit{lookup}_p\langle \hat{N}, \pi, \mathit{freeze}_N CE\rangle = [\iota; \hat{\iota}, n_1 \mapsto \pi \ldots n_k \mapsto \pi; \overline{\mu} \mid n] \\
\qquad \text{if } \hat{N} \neq \pi, N \notin \mathit{img}(\iota), \\
\qquad \mathit{lookup}_p\langle \hat{N}, \pi.1, CE\rangle = [\iota, n_1 \mapsto N \ldots n_k \mapsto N; \hat{\iota}; \overline{\mu} \mid n] \\
\mathit{lookup}_p\langle \pi, \pi, \mathit{freeze}_N CE\rangle = [\iota; \hat{\iota}; \overline{\mu}[n/n_1] \ldots [n/n_k] \mid n] \\
\qquad \text{if } N \notin \mathit{img}(\iota), \\
\qquad \mathit{lookup}_p\langle N, \pi.1, CE\rangle = [\iota, n_1 \mapsto N \ldots n_k \mapsto N; \hat{\iota}; \overline{\mu} \mid n] \\
\mathit{lookup}_p\langle \hat{N}, \pi, CE[(\overline{C\ x})\{\bar{e}\}]\rangle = \mathit{lookup}_p\langle \hat{N}, \pi.1, CE\rangle \\
\\
\mathit{k\text{-}lookup}_p(C) = \mathit{k\text{-}lookup}_p(CE) \\
\qquad \text{if } p(C) = CE \\
\mathit{k\text{-}lookup}_p([\emptyset \mid o \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}]) = \kappa \\
\mathit{k\text{-}lookup}_p(CE_1 + CE_2) = (\overline{C\ x})\{\alpha_1(\phi_1), \alpha_2(\phi_2)\} \\
\qquad \text{if } \mathit{k\text{-}lookup}_p(CE_1) = (\overline{C\ x})\{\phi_1\}, \\
\qquad \mathit{k\text{-}lookup}_p(CE_2) = (\overline{C\ x})\{\phi_2\} \\
\mathit{k\text{-}lookup}_p({}_{\sigma^\iota|}CE_{|\sigma^o}) = \mathit{k\text{-}lookup}_p(CE) \\
\mathit{k\text{-}lookup}_p(\mathit{freeze}_N CE) = \mathit{k\text{-}lookup}_p(CE) \\
\mathit{k\text{-}lookup}_p(CE[(\overline{D\ y})\{\bar{e}\}]) = (\overline{D\ y})\{\phi[\bar{e}/\bar{x}]\} \\
\qquad \text{if } \bar{x} = x_1 \ldots x_n, \\
\qquad \mathit{k\text{-}lookup}_p(CE) = (C_1\ x_1 \ldots C_n\ x_n)\{\phi\}
\end{array}
$$

Fig. 13. Lookup and constructor lookup.

triggered (see more explanations below). In flattening semantics, C is always a basic class, hence this case never happens. A generalized block expression $[\hat{\iota}; \overline{\mu}; v \mid e]$ is well formed only if

$$\mathit{names}(e) \subseteq \mathit{dom}(\hat{\iota}) \cup \mathit{dom}(\overline{\mu}) \cup \mathit{dom}(v)$$

and these three sets are disjoint.

The second section of the figure contains the new rules for expression reduction.

When a member reference (external name or path) NT needs to be resolved, the lookup procedure starts the search of NT from receiver's class C and, if successful, returns a corresponding internal name inside a block expression, as shown in rules (CLIENT-FIELD) and (CLIENT-INVK). In flattening semantics, C is always a basic class, hence lookup is trivial and the side condition can be equivalently expressed as in the analogous rules in Fig. 3.

When an internal name n is encountered, it is either directly mapped to a definition, or to a path. The former case happens when n was a local name in the basic class containing the definition of the method which is currently being executed. In this case, the corresponding definition is taken, as shown in rules (INT-FIELD) and (INT-INVK). The latter case happens when n was the internal name of an abstract or virtual member inside the basic class containing the definition of the method which is currently executed, and n has been permanently bound to some definition by an outer freeze operator (recall that only classes where all members are frozen can be instantiated). In this case, lookup of this definition is started from receiver's class via the path π, and, if successful, the internal name n is replaced by the name n' found by lookup; moreover, the corresponding path map and methods are merged with the original ones (α-renaming can be used to avoid conflicts among internal names in this phase). This is shown in rule (PATH). In flattening semantics, the latter case never happens, hence only the first two rules are needed.

Creation of an instance of class, say, C, also involves a *constructor lookup* procedure, which returns, starting from class C, the appropriate constructor, by retrieving and possibly modifying constructors of other classes (this generalizes what happens in standard Java-like languages, where the superclass constructor is always invoked). In flattening semantics, C is always a basic class, hence constructor lookup is trivial and the side condition can be equivalently expressed as in the corresponding rule in Fig. 3.

The remaining rule is analogous to that given for the flattening case.

Lookup and constructor lookup are defined in Fig. 13.

The lookup procedure is modeled by a function which, given a program p, takes three more arguments: a member reference (external name or path) NT, a path π, which acts as an accumulator and keeps track of the current subterm of the class expression which is examined, and a class name C. When lookup is started, π is always the empty path Λ, and $lookup_p\langle \hat{N}, \Lambda, C\rangle$ is abbreviated by $lookup_p\langle \hat{N}, C\rangle$.

The lookup function returns a 4-tuple consisting of input map, path map, methods and an internal name, written $[\iota; \hat{\iota}; \overline{\mu} \mid n]$. However, the final result of lookup (that is, the result returned for the initial call) is expected to be always of form $[\emptyset; \hat{\iota}; \overline{\mu} \mid n]$, abbreviated by $[\hat{\iota}; \overline{\mu} \mid n]$, since all internal names of abstract/virtual members are expected to be eventually bound to a path as effect of some freeze operator.

The first two clauses defining lookup are trivial and state that looking for a member reference starting from a class name C means looking in the definition of C, and that looking for an external name N in a basic class only succeeds if the name is present in the class, and returns the corresponding input map, methods and internal name. Note that the case where we look for a path π in a basic class is expected to never happen.

The third clause defines lookup on a sum expression. In this case, lookup is propagated to both arguments. Since we are defining a function, this means that the result is undefined if two different results are obtained. However, this is expected not to happen on class expressions which can be safely flattened, since in this case an external name cannot be found on both sides. For member references which are paths, instead, determinism is guaranteed by construction since the path exactly corresponds to a subterm. In case lookup succeeds on one of the two arguments, the result is modified by renaming field local names in a way which keeps track of this argument. For instance, if lookup succeeded on the first argument, then every field internal name f is renamed to $f.1$. This renaming is denoted by α_i. We choose this canonical α-renaming for concreteness, but any other could be chosen, provided that it is consistent with that in constructor lookup.

The fourth clause defines lookup on a reduct expression. In this case, lookup of an external name is propagated under the name the member has in the argument, given by the output renaming σ^o. Instead, lookup of a path is simply propagated, since paths are permanent references which are not affected by renamings. Moreover, the result of lookup on the argument must be modified to ensure that internal names refer to the appropriate external names obtained via the input renaming σ^ι.

There are two clauses defining lookup on a freeze expression. The former handles most cases, except the special situation in which we are exactly looking for the member that has been frozen in the current subterm π, which has the form $freeze_N CE$. In this special case (second clause) the lookup of N in CE is triggered. Moreover, the result is modified, since internal names referring to N must now refer to the permanent reference π. Otherwise (first clause), the lookup is propagated, and the result of the lookup on the argument is modified as in the previous case.

We prove now that flattening is equivalent to direct semantics (Theorem 17). To this end, we first of all define an equivalence relation on triples $\langle p, C, \lambda\rangle$, and a corresponding congruence relation on pairs $\langle p, e\rangle$.

Definition 14.

(1) Let $\sim$ be the least equivalence relation on triples $\langle p, C, \lambda\rangle$ such that:
A: $\langle p, C, [\iota; \hat{\iota}, n_1 \mapsto \pi, \ldots n_k \mapsto \pi; \overline{\mu} \mid n]\rangle \sim \langle p, C, [\iota; \hat{\iota}', \hat{\iota}; \overline{\mu}', \overline{\mu}[n'/n_1]\ldots[n'/n_k] \mid n]\rangle$ if $lookup_p\langle C, \pi\rangle = [\hat{\iota}'; \overline{\mu}' \mid n']$.
B: $\langle p, C, [\iota; \hat{\iota}; \overline{\mu}, \mu \mid n]\rangle \sim \langle p, C, [\iota; \hat{\iota}; \overline{\mu} \mid n]\rangle$ if $mu = C\ m(\overline{C\,x})\{\texttt{return}\ e;\}$, $m \notin (names(\overline{\mu}) \cup \{n\})$.
C: $\langle p, C, [\iota; \emptyset; \overline{\mu} \mid n]\rangle \sim \langle p', C', [\iota; \emptyset; \overline{\mu} \mid n]\rangle$.
(2) Let $\sim$ be the least congruence relation on pairs $\langle p, e\rangle$ such that, if $\langle p, C, [\hat{\iota}; \overline{\mu} \mid n]\rangle \sim \langle p', C, [\hat{\iota}'; \overline{\mu}' \mid n]\rangle$, then
D: $\langle p, [\hat{\iota}; \overline{\mu}; v^C \mid \mathcal{E}\{f\}]\rangle \sim \langle p', [\hat{\iota}'; \overline{\mu}'; v^C \mid \mathcal{E}\{f\}]\rangle$,
E: $\langle p, [\hat{\iota}; \overline{\mu}; v^C \mid \mathcal{E}\{m(\overline{v})\}]\rangle \sim \langle p', [\hat{\iota}'; \overline{\mu}'; v^C \mid \mathcal{E}\{m(\overline{v})\}]\rangle$.

Clause (A) states that a lookup result is equivalent to another where associations from internal names to a given path have been resolved by lookup, and path map and methods expanded. The lookup result on the left-hand side, intuitively, is a lazy version which requires a further lookup of π only when some n_i is needed, whereas in the right-hand side this lookup has already been performed. Clause (B) states that a lookup result is equivalent to another where a useless method has been removed. Finally, clause (C) states that a lookup result with no paths is no longer dependent on a program and class name. Clauses (D) and (E) state that expressions obtained via equivalent lookup results are equivalent.

Lemma 15. *If $p \longrightarrow p'$ then, for each N, C,*

(1) *$lookup_p\langle N, C\rangle$ and $lookup_{p'}\langle N, C\rangle$ are either both defined as λ, λ' respectively, and $\langle p, C, \lambda\rangle \sim \langle p', C, \lambda'\rangle$, or both undefined,*
(2) *k-$lookup_p(C)$ and k-$lookup_{p'}(C)$ are both defined and equal, or both undefined.*

Proof. By induction on the definition of $p \longrightarrow p'$.

(CDEC1) We have

$$CE \longrightarrow CE',$$
$$p \equiv (p_1, C \mapsto CE) \longrightarrow p' \equiv (p_1, C \mapsto CE').$$

We show, by induction on the definition of $CE \longrightarrow CE'$, that, for all N, π, if CE is the π-subterm of $p(C)$:

(1) $lookup_p\langle N, \pi, CE\rangle$ and $lookup_{p'}\langle N, \pi, CE'\rangle$ are either both defined as λ, λ' respectively, and $\langle p, C, \lambda\rangle \sim \langle p', C, \lambda'\rangle$, or both undefined,
(2) $k\text{-}lookup_p(CE)$ and $k\text{-}lookup_{p'}(CE')$ are both defined and equal, or both undefined.
This is enough to prove the thesis since other class names are not affected.

(SUM) We have

$$\begin{aligned}
&CE \equiv CE_1 + CE_2,\\
&CE_1 \equiv [\iota_1 \mid o_1 \mid I\ \{\overline{\varphi}_1\ (\overline{C\,x})\{\phi_1\}\ \overline{\mu}_1\}],\\
&CE_2 \equiv [\iota_2 \mid o_2 \mid I\ \{\overline{\varphi}_2\ (\overline{C\,x})\{\phi_2\}\ \overline{\mu}_2\}],\\
&CE' \equiv [\alpha_1(\iota_1), \alpha_2(\iota_2) \mid \alpha_1(o_1), \alpha_2(o_2) \mid\\
&\quad I\ \{\alpha_1(\overline{\varphi}_1), \alpha_2(\overline{\varphi}_2)\ (\overline{C\,x})\{\alpha_1(\phi_1), \alpha_2(\phi_2)\}\ \alpha_1(\overline{\mu}_1), \alpha_2(\overline{\mu}_2)\}],
\end{aligned}$$

where, in applying rule (SUM), we have arbitrarily chosen to rename field local names by the canonical renamings α_1 and α_2.

(1) $lookup_p\langle N, \pi, CE\rangle$ and $lookup_{p'}\langle N, \pi, CE'\rangle$ are both defined only if $(o_1, o_2)(N) = n$ for some n. By well-formedness of o_1, o_2 this means that either $o_1(N)$ is defined or $o_2(N)$ is defined, but not both. Let us assume $o_1(N) = n$ (the other case is analogous). Then,

$$\begin{aligned}
&lookup_p\langle N, \pi, CE\rangle = \lambda \equiv [\alpha_1(\iota_1); \emptyset; \alpha_1(\overline{\mu}_1) \mid \alpha_1(n)],\\
&lookup_{p'}\langle N, \pi, CE'\rangle = \lambda' \equiv [\alpha_1(\iota_1), \alpha_2(\iota_2); \emptyset; \alpha_1(\overline{\mu}_1), \alpha_2(\overline{\mu}_2) \mid \alpha_1(n)].
\end{aligned}$$

We get the thesis since, by well-formedness conditions, $\alpha_1(n) \in dom(\alpha_1(\iota_1)) \cup dom(\alpha_1(\overline{\mu}_1))$ and $names(\alpha_1(\overline{\mu}_1)) \cap (dom(\alpha_2(\iota_2)) \cup dom(\alpha_2(\overline{\mu}_2)) = \emptyset$, hence $\langle p, C, \lambda\rangle \sim \langle p', C, \lambda'\rangle$ by clauses (B) and (C) in Definition 14.

(2) $k\text{-}lookup_p(CE)$ and $k\text{-}lookup_{p'}(CE')$ are both defined and equal to $(\overline{C\,x})\{\alpha_1(\phi_1), \alpha_2(\phi_2)\}$.

(REDUCT) We have

$$\begin{aligned}
&CE \equiv {}_{\sigma^\iota|}[\iota \mid o \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}]_{|\sigma^o},\\
&CE' \equiv [\sigma^\iota \circ \iota \mid o \circ \sigma^o \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}].
\end{aligned}$$

(1) $lookup_p\langle N, \pi, CE\rangle$ and $lookup_{p'}\langle N, \pi, CE'\rangle$ are both defined only if $o(\sigma^o(N)) = n$ for some n, and in this case they are both equal to $[\sigma^\iota \circ \iota; \emptyset; \overline{\mu} \mid n]$.
(2) Trivial.

(FREEZE) We have

$$\begin{aligned}
&CE \equiv freeze_N[\iota, n_1 : T \mapsto N \ldots n_k : T \mapsto N \mid o \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}\}],\\
&CE' \equiv [\iota \mid o \mid I\ \{\overline{\varphi}\ \kappa\ \overline{\mu}[n/n_1]\ldots[n/n_k]\}],\\
&N \notin img(\iota), o(N) = n.
\end{aligned}$$

(1) $lookup_p\langle N', \pi, CE\rangle$ and $lookup_p\langle N', \pi, CE'\rangle$ are defined only if $o(N') = n'$ for some n'. Then,

$$\begin{aligned}
&lookup_p\langle N', \pi, CE\rangle = \lambda \equiv [\iota; n_1 \mapsto \pi \ldots n_k \mapsto \pi; \overline{\mu} \mid n'],\\
&lookup_p\langle N', \pi, CE'\rangle = \lambda' \equiv [\iota; \emptyset; \overline{\mu}[n/n_1]\ldots[n/n_k] \mid n'].
\end{aligned}$$

Since CE is the π-subterm of $p(C)$, $lookup_p\langle \pi, C\rangle = lookup_p\langle \pi, \pi, CE\rangle = [\iota; \emptyset; \overline{\mu}[n/n_1]\ldots[n/n_k] \mid n']$, hence $\langle p, C, \lambda\rangle \sim \langle p', C, \lambda'\rangle$ by clauses (A) and (C) in Definition 14.

(2) Trivial.

(CONSTRUCTOR WRAPPING) Trivial.

($\mathcal{CE}$) The proof is by structural induction on the context. We show the following case (the others are analogous):

$$\begin{aligned}
&CE \equiv CE_1 + CE_2,\\
&CE' \equiv CE'_1 + CE_2,\\
&CE_1 \longrightarrow CE'_1.
\end{aligned}$$

(1) By inductive hypothesis, $lookup_p\langle N, \pi.1, CE_1\rangle$ and $lookup_p\langle N, \pi.1, CE'_1\rangle$ are either both defined as λ, λ', respectively, and $\langle p, C, \lambda\rangle \sim \langle p, C, \lambda'\rangle$, or both undefined.
- Assume the former case holds and $lookup_p\langle N, \pi.2, CE_2\rangle$ is undefined. Then, $lookup_p\langle N, \pi, CE\rangle$ and $lookup_{p'}\langle N, \pi, CE'\rangle$ are (uniquely) defined as λ, λ', respectively, and we get the thesis.
- Assume the former case holds and $lookup_p\langle N, \pi.2, CE_2\rangle$ is defined. Then, $lookup_p\langle N, \pi, CE\rangle$ and $lookup_{p'}\langle N, \pi, CE'\rangle$ are both undefined.
- Assume the latter case holds. Then, $lookup_p\langle N, \pi, CE\rangle$ and $lookup_{p'}\langle N, \pi, CE'\rangle$ are both equal to $lookup_p\langle N, \pi.2, CE_2\rangle$.

(2) Trivial by inductive hypothesis.

(CDEC2) We have

$$(p, C \mapsto B) \longrightarrow (p[B/C], C \mapsto B)$$

(1) and (2) can be easily proved by induction on the definition of $p[B/C]$. □

Then, Theorem 17 follows as a corollary of the following, where we write $e \xrightarrow{0/1}_p e'$ to denote that either $e \longrightarrow_p e'$ or $e \equiv e'$.

Theorem 16. *If $p \longrightarrow p'$ and $\langle p, e_1\rangle \sim \langle p', e_2\rangle$, then $e_1 \longrightarrow_p e_1'$ implies that $e_2 \xrightarrow{0/1}_{p'} e_2'$ with $\langle p, e_1'\rangle \sim \langle p', e_2'\rangle$, and conversely.*

Proof. We prove the direct implication by induction on the definition of $e_1 \longrightarrow_p e_1'$. The converse implication can be proved analogously by induction on the definition of $e_2 \longrightarrow_p e_2'$.

(INT-FIELD), (INT-INVK), (EXIT-BLOCK) Reduction does not depend on the program, hence the thesis trivially holds.
(CLIENT-FIELD) We have

$$e_1 \equiv v^C.M(\overline{v}) \longrightarrow_p [\hat{\imath}; \overline{\mu}; v^C \mid m(\overline{v})],$$
$$\mathit{lookup}_p\langle M, C\rangle = \lambda \equiv [\hat{\imath}; \overline{\mu} \mid m].$$

By definition of $\sim$-congruence, e_1' can only coincide with e_1. By Lemma 15(1), $\mathit{lookup}_{p'}\langle M, C\rangle = \lambda' \equiv [\hat{\imath}'; \overline{\mu}' \mid m]$ and $\langle p, C, \lambda\rangle \sim \langle p', C, \lambda'\rangle$. Hence,

$$v^C.M(\overline{v}) \longrightarrow_{p'} [\hat{\imath}'; \overline{\mu}'; v^C \mid m(\overline{v})],$$

and the thesis follows by Definition 14(2, D).
(CLIENT-INVK) Analogously to the above, by Definition 14(2, E).
(PATH) We have

$$e_1 \equiv [\hat{\imath}, n \mapsto \pi; \overline{\mu}; v^C \mid e] \longrightarrow_p e_2 \equiv [\hat{\imath}, \hat{\imath}'; \overline{\mu}[n'/n], \overline{\mu}'; v^C \mid e[n'/n]],$$
$$\mathit{lookup}_p\langle \pi, C\rangle = \lambda \equiv [\hat{\imath}'; \overline{\mu}' \mid n'].$$

By Definition 14(1, A), $\langle p, e_1\rangle \sim \langle p, e_2\rangle$, hence, since $\langle p, e_1\rangle \sim \langle p', e_1'\rangle$, by transitivity $\langle p', e_1'\rangle \sim \langle p, e_2\rangle$, and we get the thesis (with a $\xrightarrow{0}$ step).
(OBJ-CREATION) We have

$$e_1 \equiv \mathtt{new}\, C(\overline{v}) \longrightarrow_p C(\overline{f = e[\overline{v}/\overline{x}]}),$$
$$\mathit{k\text{-}lookup}_p(C) = (C_1\, x_1 \ldots C_n\, x_n)\{\phi\},$$
$$\overline{x} = x_1 \ldots x_n.$$

By definition of $\sim$-congruence, e_1' can only coincide with e_1. By Lemma 15(2), $\mathit{k\text{-}lookup}_{p'}(C) = \mathit{k\text{-}lookup}_p(C)$, and we get the thesis.
($\mathcal{E}$) The thesis follows from the fact that $\sim$ is a congruence. □

We denote by $\xrightarrow{\star}$ the reflexive and transitive closure of the flattening relation, and analogously for the reduction relation.

Theorem 17. *If $p \xrightarrow{\star} p'$, and e is an expression with no paths, then $e \xrightarrow{\star}_p v$ iff $e \xrightarrow{\star}_{p'} v$.*

Proof. The fact that, if $p \longrightarrow p'$, and e is an expression with no paths, then $e \xrightarrow{\star}_p v$ iff $e \xrightarrow{\star}_{p'} v$ follows from the two implications of Theorem 16 by induction on the number of steps in $e \xrightarrow{\star}_p v$ and $e \xrightarrow{\star}_{p'} v$, respectively. Indeed, since e has no paths, $\langle p, e\rangle \sim \langle p', e\rangle$ by Definition 14(1, C), and two $\sim$-equivalent values can only coincide.

Then, the thesis follows by induction on the number of steps in $p \xrightarrow{\star} p'$. □

5. Conclusion

We have presented FJIG, a core calculus which formalizes the Bracha's Jigsaw framework [1] in a Java-like setting. The design of FJIG comes out naturally, yet not trivially, by taking Featherweight Java [13] as starting point and replacing inheritance by the more general composition operators of Jigsaw.

We believe that such a core calculus can be useful for many research directions. First, it provides a simple unifying formalism for encoding and comparing a large variety of different mechanisms for software composition in class-based

languages, including standard inheritance, mixin classes, traits and hiding. Then, it can serve as the basis for the design of a real language based on Jigsaw principles. Moreover, it could be enriched by behavioural types, leading to a class-based specification language, in the spirit of, e.g., JML [20], allowing modular development and composition of class specifications.

We have also defined two different execution models for the calculus, flattening and direct semantics, and proved their equivalence. That is, we have shown the equivalence of two different views on inheritance in a formal setting with a more sophisticated composition mechanism, where, e.g., mixin classes and traits can be subsumed. This can also greatly help in integrating such features, or other modularity mechanisms, in standard class-based languages, since it gives practical hints on implementation.

Many proposals for extending the object-oriented paradigm have just taken one approach or the other. In particular, the most direct source of inspiration for our work has been [11], which defines a direct semantics for traits. Essentially, their dynamic look-up algorithm can be seen as a simplified version, handling sum and output reduct only, of ours. On the other hand, to the best of our knowledge there has been no attempt at providing both semantics and proving their equivalence, as we do in this paper, for any of these extensions, with the exception of [21], where the direct semantic for traits provided by [11] is proved equivalent to a flattening counterpart in the tradition of [9].

Apart from the two key references mentioned above, this work has been directly influenced by work on traits [9,10], mostly by the recent developments [11,22,23]. In particular, we share with [22,23] the objective of replacing inheritance by more flexible operators, but taking the exact opposite approach: instead of splitting the three roles of classes (module, object generator and type) into three different constructs (trait, class and interface), here we take classes as modules, as in the original Jigsaw framework, and we also keep the nominal approach of Java-like languages where all class names are types.

Since the initial submission of this paper, we have developed extensions of FJig in two orthogonal directions. In [24–26] we have added a *meta-level* allowing the programmer to define her/his own composition operators. In [19,27] we have added nested classes, making composition operators *deep*, similarly to *deep mixin composition* [28,29] and *family polymorphism* [30–33], which, however, only consider an asymmetric sum operator. A notable exception is the operator & of J& [34], an extension of [33] with a symmetric composition mechanism similar to our sum operator. The extension of FJig with nested classes allows one to easily encode generics.

These two extensions have been integrated in Marco Servetto's PhD thesis [35].

Other interesting topics to be possibly investigated are smart implementation techniques of direct semantics, and equivalence between flattening and direct semantics in languages allowing features whose runtime behaviour depends on static types, such as overloading or static binding of members.

A very preliminary interpreter of FJig flattening semantics, assigned as master thesis, can be found at http://www.disi.unige.it/person/LagorioG/FJig/. A more worked out prototype, including real-language features such as primitive types, predefined classes, `void` methods, and some statements, has been developed for the meta-level extension, see http://www.disi.unige.it/person/ServettoM/MetaFJig/.

Acknowledgments

We warmly thank the anonymous referees of FOOL'09, ECOOP'09 and this journal for many helpful comments.

References

[1] G. Bracha, The programming language JIGSAW: Mixins, modularity and multiple inheritance, PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
[2] J.B. Wells, R. Vestergaard, Confluent equational reasoning for linking with first-class primitive modules, in: ESOP 2000 – European Symposium on Programming, in: Lecture Notes in Comput. Sci., vol. 1782, Springer, 2000, pp. 412–428.
[3] D. Ancona, E. Zucca, A calculus of module systems, J. Funct. Programming 12 (2) (2002) 91–132.
[4] D. Duggan, C. Sourelis, Mixin modules, in: Intl. Conf. on Functional Programming 1996, ACM Press, 1996, pp. 262–273.
[5] T. Hirschowitz, X. Leroy, Mixin modules in a call-by-value setting, in: D. Le Métayer (Ed.), ESOP 2002 – European Symposium on Programming 2002, in: Lecture Notes in Comput. Sci., vol. 2305, Springer, 2002, pp. 6–20.
[6] T. Hirschowitz, X. Leroy, J.B. Wells, Call-by-value mixin modules: Reduction semantics, side effects, types, in: D.A. Schmidt (Ed.), ESOP 2003 – European Symposium on Programming 2003, in: Lecture Notes in Comput. Sci., vol. 2986, Springer, 2004, pp. 64–78.
[7] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and mixins, in: ACM Symp. on Principles of Programming Languages 1998, ACM Press, 1998, pp. 171–183.
[8] D. Ancona, G. Lagorio, E. Zucca, Jam–designing a Java extension with mixins, ACM Trans. Progr. Lang. Syst. 25 (5) (2003) 641–712, extended version of [36].
[9] N. Schärli, S. Ducasse, O. Nierstrasz, A.P. Black, Traits: Composable units of behaviour, in: ECOOP'03 – Object-Oriented Programming, in: Lecture Notes in Comput. Sci., vol. 2743, Springer, 2003, pp. 248–274.
[10] K. Fisher, J. Reppy, A typed calculus of traits, in: FOOL'04 – Intl. Workshop on Foundations of Object-Oriented Languages, 2004.
[11] L. Liquori, A. Spiwack, FeatherTrait: A modest extension of Featherweight Java, ACM Trans. Progr. Lang. Syst. 30 (2) (2008) 1–32.
[12] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Stateful traits, in: Advances in Smalltalk – 14th International Smalltalk Conference, vol. 4406, ISC, 2006, Springer, 2007, pp. 66–90.
[13] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Trans. Progr. Lang. Syst. 23 (3) (2001) 396–450.
[14] G. Lagorio, M. Servetto, E. Zucca, Flattening versus direct semantics for Featherweight Jigsaw, in: FOOL'09 – Intl. Workshop on Foundations of Object-Oriented Languages, 2009.
[15] G. Lagorio, M. Servetto, E. Zucca, Featherweight Jigsaw – A minimal core calculus for modular composition of classes, in: S. Drossopoulou (Ed.), ECOOP'09 – Object-Oriented Programming, in: Lecture Notes in Comput. Sci., vol. 5653, Springer, 2009.

[16] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Stateful traits and their formalization, Comput. Lang. Syst. Struct. 34 (2–3) (2008) 83–108.
[17] D. Ancona, E. Zucca, Overriding operators in a mixin-based framework, in: H. Glaser, P. Hartel, H. Kuchen (Eds.), PLILP '97 – 9th Intl. Symp. on Programming Languages, Implementations, Logics and Programs, in: Lecture Notes in Comput. Sci., vol. 1292, Springer, 1997, pp. 47–61.
[18] E. Gamma, R. Helm, R.E. Johnson, J.M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison–Wesley Professional Computing Series, Addison–Wesley, 1995.
[19] A. Corradi, M. Servetto, E. Zucca, DeepFJig – Modular composition of nested classes, in: FOOL 2010 – Intl. Workshop on Foundations of Object-Oriented Languages, 2010.
[20] G.T. Leavens, Tutorial on JML, the Java modeling language, in: R.E.K. Stirewalt, A. Egyed, B. Fischer (Eds.), Automated Software Engineering, ASE, 2007, ACM Press, 2007.
[21] O. Nierstrasz, S. Ducasse, N. Schärli, Flattening traits, J. Object Technol. 5 (2006) 66–90.
[22] V. Bono, F. Damiani, E. Giachino, Separating type, behavior, and state to achieve very fine-grained reuse, in: 9th Intl. Workshop on Formal Techniques for Java-like Programs, 2007.
[23] V. Bono, F. Damiani, E. Giachino, On traits and types in a Java-like setting, in: TCS'08 – IFIP Int. Conf. on Theoretical Computer Science, Springer, 2008.
[24] G. Lagorio, M. Servetto, E. Zucca, Customizable composition operators for Java-like classes (extended abstract), in: ICTCS'09 – Italian Conf. on Theoretical Computer Science, 2009.
[25] G. Lagorio, M. Servetto, E. Zucca, A lightweight approach to customizable composition operators for Java-like classes, in: FACS'09 – International Workshop on Formal Aspects of Component Software, in: Electron. Notes Theor. Comput. Sci., vol. 263, 2010, pp. 161–177.
[26] M. Servetto, E. Zucca, MetaFJig – A meta-circular composition language for Java-like classes, in: W.R. Cook, S. Clarke, M.C. Rinard (Eds.), ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 2010, ACM Press, 2010, pp. 464–483.
[27] A. Corradi, M. Servetto, E. Zucca, DeepFJig – Modular composition of nested classes, in: C. Wimmer, C.W. Probst (Eds.), PPPJ'11 – Principles and Practice of Programming in Java, in: ACM Internat. Proc. Ser., ACM Press, 2011, pp. 101–110.
[28] M. Odersky, M. Zenger, Independently extensible solutions to the expression problem, in: FOOL'05 – Intl. Workshop on Foundations of Object-Oriented Languages, 2005.
[29] D. Hutchins, Eliminating distinctions of class: Using prototypes to model virtual classes, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, 2006, ACM Press, 2006, pp. 1–20.
[30] E. Ernst, K. Ostermann, W.R. Cook, A virtual class calculus, in: J.G. Morrisett, S.L.P. Jones (Eds.), ACM Symp. on Principles of Programming Languages 2006, vol. 41, ACM Press, 2006, pp. 270–282.
[31] A. Igarashi, C. Saito, M. Viroli, Lightweight family polymorphism, in: K. Yi (Ed.), APLAS 2005 – Asian Symposium on Programming Languages and Systems, in: Lecture Notes in Comput. Sci., vol. 3780, Springer, 2005, pp. 161–177.
[32] A. Igarashi, M. Viroli, Variant path types for scalable extensibility, in: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA, 2007, ACM Press, 2007, pp. 113–132.
[33] N. Nystrom, S. Chong, A.C. Myers, Scalable extensibility via nested inheritance, SIGPLAN Not. 39 (10) (2004) 99–115, http://doi.acm.org/10.1145/1035292.1028986.
[34] N. Nystrom, X. Qi, A.C. Myers, J&: Nested intersection for scalable software composition, SIGPLAN Not. 41 (10) (2006) 21–36, http://doi.acm.org/10.1145/1167515.1167476.
[35] M. Servetto, MetaFJig – A meta-circular composition language for Java-like classes, PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2011, http://bart.disi.unige.it/bibliography/files/servettoMarco.pdf.
[36] D. Ancona, G. Lagorio, E. Zucca, Jam: A smooth extension of Java with mixins, in: E. Bertino (Ed.), ECOOP'00 – European Conference on Object-Oriented Programming, in: Lecture Notes in Comput. Sci., vol. 1850, Springer, 2000, pp. 154–178, extended version of [8].